Spis treści

Na podstawie:

- *M.J.Rochkind - Programowanie w systemie UNIX dla zaawansowanych (Advanced UNIX Programming)*
- `https://ai.ia.agh.edu.pl/wiki/pl:dydaktyka:so:2017:labs:lab_intro`
- *Graham Glass, King Ables - Linux dla programistów i użytkowników*

1 Pojęcia podstawowe potrzebne do zrozumienia tematu

Zaawansowana komunikacja międzyprocesowa - Motywacja

Zaawansowana komunikacja międzyprocesowa (w skrócie IPC) jest potrzebna ze względu na ograniczenia jakie mają łącza nienazwane (np. konieczność tworzenia procesu potomnego do dziedziczenia deskryptorów plików). Do zaawansowanych mechanizmów komunikacji międzyprocesowej należą łącza nazwane, kolejki wiadomości, semafory, zamki plików, pamięć współdzielona, gniazda.

- Jakie mechanizmy należą do zaawansowanej komunikacji międzyprocesowej ?
- Jakie są wady łącz nienazwanych ?

2 Łącza nazwane (inaczej FIFO)

Łącza nazwane

Łącza nazwane (FIFO) to mechanizm umożliwiający komunikację pomiędzy procesami, które nie muszą być ze sobą w żaden sposób powiązane (nie muszą być powiązane relacją dziedziczenia tak jak w przypadku łącz nienazwanych). FIFO łączy w sobie cechy regularnego pliku oraz łącza nienazwanego, które było omówione w poprzednim lab. FIFO jest podobne do pliku, ponieważ można je otwierać i zamykać w celu czytania i zapisywania danych. Jest podobne do łącz nienazwanych, ponieważ zapisywanie i odczytywanie odbywa się w podobny sposób.

2.1. Otwieranie i zamykanie FIFO

FIFO przy otwieraniu do czytania czeka aż zostanie otwarte do zapisywania. Przy otwieraniu do zapisywania czeka aż zostanie otwarte do czytania. Takie zachowanie umożliwia synchronizację procesów.
Jeśli flaga O_NONBLOCK jest ustawiona to open dla czytania wraca natychmiast i zwraca deskryptor pliku do czytania bez czekania na writera, a w przypadku open dla zapisywania zwraca -1 jeśli żaden reader nie jest otworzony.
Do otwierania FIFO do odczytu i zapisu używamy funkcji open (przykłady w dalszej części).

- Czym jest FIFO ?
- Jak zachowuje się FIFO przy zamykaniu/otwieraniu ? Jaki wpływ ma flaga O_NONBLOCK ?

3 Funkcja mkfifo - Tworzenie FIFO

FIFO nie tworzy się przy pomocy wywołania open, tworzymy je przy pomocy wywołania systemowego *mkfifo*.

```
// mkfifo - make FIFO
#include <sys/stat.h>
int mkfifo(
  const char *path, /* pathname */
  mode_t perms /* permissions */
); /* Returns 0 on success or -1 on error (sets errno) */
```

3.1. Argumenty wywołania

- const char *path - to nazwa FIFO
- mode_t perms - to argumenty określające kto będzie mógł korzystać z FIFO (tak samo jak przy otwieraniu plików)

3.2. Przykład wywołania mkfifo

```
#define SERVER_FIFO_NAME "fifo_server"

if (mkfifo(SERVER_FIFO_NAME, PERM_FILE) == -1 && errno != EEXIST)
  EC_FAIL
```

3.3. Wartość zwracana

Funkcja zwraca 0 w przypadku sukcesu oraz kod błędu w przypadku błędu.

- Do czego służy funkcja mkfifo ?
- Jakie są argumenty wywołania funkcji mkfifo ?
- Co zwraca mkfifo ?

4 Otwieranie FIFO do zapisu i odczytu przy pomocy open

Po tym jak stworzymy FIFO musimy otworzyć plik do zapisu i odczytu aby dostać deskryptory plików. W tym celu korzysta się z funkcji open. Przykłady wywołania open

```
#define SERVER_FIFO_NAME "fifo_server"

// przyklad wywolania open dla read
if (mkfifo(SERVER_FIFO_NAME, PERM_FILE) == -1 && errno != EEXIST)
  EC_FAIL
ec_negl( fd_server = open(SERVER_FIFO_NAME, O_RDONLY) ) // blokowane do czasu gdy nie ma
     otwartego writera

// przyklad wywolania open dla write
fd_client = open(fifo_name, O_WRONLY)
```

5 Przekazywanie informacji przez FIFO

Rozważmy sytuacje gdzie mamy komunikację klient-serwer, serwer może obsługiwać wiele klientów.

5.0.1. Serwer

Wtedy po stronie serwera tworzymy FIFO o wybranym imieniu i to FIFO otwieramy do czytania danych. Do wysyłania danych potrzebna nam jest jednak informacja od którego procesu przyszła wiadomość, aby do niego wysłać odpowiedź. W tym celu przekazujemy przy wysyłaniu wiadomości do serwera strukturę, która zawiera wiadomość oraz PID procesu wysyłającego.

```
struct simple_message {
  pid_t sm_clientpid; // PID procesu
  char sm_data[200];  // dane
};
```

Na podstawie przesłanego PID tworzymy sobie nazwę FIFO klienta do którego chcemy zapisać dane, np. przy użyciu funkcji:

```
bool make_fifo_name(pid_t pid, char *name, size_t name_max) {
  snprintf(name, name_max, "fifo%ld", (long)pid);
  return true;
}
```

W procesie serwera odbieramy więc dane z kolejki, którą sami tworzymy:

```
if (mkfifo(SERVER_FIFO_NAME, PERM_FILE) == -1 && errno != EEXIST)
  EC_FAIL
ec_neg1( fd_server = open(SERVER_FIFO_NAME, O_RDONLY) )
while (true) {
  ec_neg1( nread = read(fd_server, &msg, sizeof(msg)) )
  if (nread == 0) {
    errno = ENETDOWN;
    EC_FAIL
  }
  ...
```

oraz wysyłamy dane do kolejki klienta (nazwę kolejki klienta tworzymy na podstawie id, które nam przesłał):

```
...
ec_false( make_fifo_name(msg.sm_clientpid, fifo_name, sizeof(fifo_name)) )
ec_neg1( fd_client = open(fifo_name, O_WRONLY) )
ec_neg1( write(fd_client, &msg, sizeof(msg)) )
ec_neg1( close(fd_client) )
...
```

5.0.2. Klient

Po stronie klienta musimy otworzyć do zapisu FIFO serwera, a także stworzyć własne FIFO do którego serwer będzie zapisywał, a z którego klient będzie czytał dane.
Tworzymy FIFO na podstawie id klienta:

```
 printf("client %ld started\n", (long)getpid());
 msg.sm_clientpid = getpid();
 ec_false( make_fifo_name(msg.sm_clientpid, fifo_name, sizeof(fifo_name)) )
 if (mkfifo(fifo_name, PERM_FILE) == -1 && errno != EEXIST)
  EC_FAIL
 ...
```

Klient otwiera FIFO serwera i zapisuje dane:

```
ec_neg1( fd_server = open(SERVER_FIFO_NAME, O_WRONLY) )
for (i = 0; work[i] != NULL; i++) {
  strcpy(msg.sm_data, work[i]);
  ec_neg1( write(fd_server, &msg, sizeof(msg))
```

Klient otwiera swoje FIFO do czytania danych:

```
if (fd_client == -1)
  ec_neg1( fd_client = open(fifo_name, O_RDONLY) )
  ec_neg1( nread = read(fd_client, &msg, sizeof(msg)) )
if (nread == 0) {
  errno = ENETDOWN;
  EC_FAIL
}
```

- Jak przekazywane są wiadomości w schemacie klient-serwer ? Gdzie stworzyć FIFO i co będzie z nim robione z którego procesu ?

6 Kod całego serwera i klienta

6.1. Kod serwera

```

int main(void) {
  int fd_server, fd_client, i;
  ssize_t nread;
  struct simple_message msg;
  char fifo_name[100];

  printf("server started\n");
  if (mkfifo(SERVER_FIFO_NAME, PERM_FILE) == -1 && errno != EEXIST)
    EC_FAIL
  ec_neg1( fd_server = open(SERVER_FIFO_NAME, O_RDONLY) )
  while (true) {
    ec_neg1( nread = read(fd_server, &msg, sizeof(msg)) ) // serwer czyta ze swojego
    FIFO
    if (nread == 0) {
      errno = ENETDOWN;
      EC_FAIL
    }
    for (i = 0; msg.sm_data[i] != '\0'; i++)
      msg.sm_data[i] = toupper(msg.sm_data[i]);
    ec_false( make_fifo_name(msg.sm_clientpid, fifo_name, sizeof(fifo_name)) ) // serwer
     zapisuje do FIFO klienta
    ec_neg1( fd_client = open(fifo_name, O_WRONLY) )
    ec_neg1( write(fd_client, &msg, sizeof(msg)) )
    ec_neg1( close(fd_client) ) // SERWER ZAMYKA KLIENTA DO ZAPISYWANIA
  }
  /* never actually get here */
  ec_neg1( close(fd_server) )
  exit(EXIT_SUCCESS);

  EC_CLEANUP_BGN
```

```
31      exit(EXIT_FAILURE);
32    EC_CLEANUP_END
33  }
```

6.2. Kod klienta

```
1  int main(int argc, char *argv[]) {
2    int fd_server, fd_client = -1, i;
3    ssize_t nread;
4    struct simple_message msg;
5    char fifo_name[100];
6    char *work[] = {
7      "applesauce",
8      "tiger",
9      "mountain",
10     NULL     };
11
12     printf("client %ld started\n", (long)getpid());
13     msg.sm_clientpid = getpid();
14     ec_false( make_fifo_name(msg.sm_clientpid, fifo_name, sizeof(fifo_name)) )
15     if (mkfifo(fifo_name, PERM_FILE) == -1 && errno != EEXIST)
16       EC_FAIL
17     ec_neg1( fd_server = open(SERVER_FIFO_NAME, O_WRONLY) )
18     for (i = 0; work[i] != NULL; i++) {
19       strcpy(msg.sm_data, work[i]);
20     ec_neg1( write(fd_server, &msg, sizeof(msg)) )
21     if (fd_client == -1)
22       ec_neg1( fd_client = open(fifo_name, O_RDONLY) )
23     ec_neg1( nread = read(fd_client, &msg, sizeof(msg)) )
24     if (nread == 0) {
25       errno = ENETDOWN;
26       EC_FAIL
27     }
28     printf("client %ld: %s --> %s\n", (long)getpid(), work[i], msg.sm_data);
29     }
30     ec_neg1( close(fd_server) )
31     ec_neg1( close(fd_client) )
32     ec_neg1( unlink(fifo_name) )
33     printf("Client %ld done\n", (long)getpid());
34     exit(EXIT_SUCCESS);
35
36   EC_CLEANUP_BGN
37     exit(EXIT_FAILURE);
38   EC_CLEANUP_END
39 }
```

7 Problemy z implementacją

Implementacja jak w kodach powyżej ma jednak bugi. Chodzi o to, że serwer po odczytaniu wiadomości ze swojego FIFO zapisuje dane do FIFO klienta i zamyka to FIFO. Później, gdy klient chce odczytać przesłane dane nie może tego zrobić, bo read zwraca EOF (deskryptor zapisujący FIFO został zamknięty). Analogiczny problem jest w przypadku serwera. Klient zamyka zapisujące FIFO i serwer dostaje EOF o ile nie ma innych klientów.
Rozwiązaniem tego problemu jest utrzymywanie deskryptora zapisującego cały czas otwartego przez klienta i utrzymywanie deskryptora zapisującego cały czas otwartego przez serwer. A więc gdy klient i serwer otwierają swoje FIFO w swoich procesach dla czytania powinny też otwierać swoje FIFO do zapisywania po to aby utrzymywać je otwarte.

- Jaki jest problem z komunikacją klient-serwer ? Jak go łatwo rozwiązać ?

8 System V i POSIX - ogólnie

System V
System V jest to zbiór wywołań systemowych służących do obsługi komunikacji międzyprocesowej (IPC), semaforów, pamięci dzielonej. Jest to system starszy od POSIX.

POSIX
POSIX to również zbiór wywołań systemowych służących do obsługi komunikacji międzyprocesowej, semaforów, pamięci dzielonej, ale jest on nowszy od System V.

- Czym są system V i POSIX ?

9 System V

9.1. ftok i identyfikatory

W system V każdy obiekt (semafor, pamięć współdzielona, kolejka wiadomości) ma przypisany swój identyfikator (inaczej klucz) aby można się było za jego pomocą odwołać do tego obiektu. Do wygenerowania klucza służy wywołanie funkcji ftok.

```
ftok-generate System V IPC key
#include <sys/ipc.h>
key_t ftok(
  const char *path, /* pathname of existing file */
  int id /* desired key number */
); /* Returns key on success or -1 on error (sets errno) */
```

Pierwszym argumentem wywołania jest ścieżka do pliku, który musi już istnieć. Na podstawie tej ścieżki tworzony jest klucz. Drugi argument to preferowany klucz, może to być jakakolwiek litera. Dzięki temu można dla jednej ścieżki wygenerować kilka różnych kluczy.

- Do czego służy wywołanie funkcji ftok ?

9.2. msgget

To funkcja służąca do tworzenia kolejek komunikatów w System V. Pierwszym jej argumentem jest klucz do obiektu. Aby stworzyć kolejkę drugi argument musi mieć ustawioną flagę O_CREAT.

```
msgget-get message-queue identifier
#include <sys/msg.h>
int msgget(
  key_t key, /* key */
  int flags /* creation flags */
); /* Returns identifier or -1 on error (sets errno) */
```

9.3. msgctl

To funkcja służąca do ustawiania parametrów stworzonej kolejki komunikatów, jak np. pozwolenia, maksymalna liczba bajtów itp.

```
#include <sys/msg.h>
int msgctl(
  int msqid, /* identifier */
  int cmd, /* command */
  struct msqid_ds *data /* data for command */
); /* Returns 0 on success or -1 on error (sets errno) */
```

- Do czego służy funkcja msgget ?
- Do czego służy funkcji msgctl ?

9.4. msgsnd i msgrcv

To funkcje służące do wysyłania i odbierania wiadomości z kolejki komunikatów.

```
msgsnd-send message
#include <sys/msg.h>
int msgsnd(
  int msqid, /* identifier */
  const void *msgp, /* message */
  size_t msgsize, /* size of message */
  int flags /* flags */
); /* Returns 0 on success or -1 on error (sets errno) */
```

```
msgrcv-receive message
#include <sys/msg.h>
ssize_t msgrcv(
  int msqid, /* identifier */
  void *msgp, /* message */
  size_t mtextsize, /* size of mtext buffer */
  long msgtype, /* message type requested */
  int flags /* flags */
); /* Returns number of bytes placed in mtext or -1 on error (sets errno) */
```

- Do czego służy funkcja msgsnd ?
- Do czego służy funkcja msgrcv ?

10 POSIX IPC

Mesage queues
POSIX IPC korzysta z tzw. message-queues (czyli kolejek wiadomości), służą one do wymiany danych pomiędzy procesami.

POSIX IPC names
POSIX IPC names to identyfikatory kolejek wiadomości w systemie. Są to po prostu nazwy (strings), które powinny zaczynać się od slash. Nazwy zaczynające się od slash jednoznacznie identyfikują message queue.

- Czym są message queues w POSIX IPC ?
- Czym są nazwy w POSIX ?

10.1. mq_open

To funkcja, która służy do utworzenia kolejki wiadomości.

```
#include <mqueue.h>
mqd_t mq_open(
  const char *name, /* POSIX IPC name */
  int flags /* flags (excluding O_CREAT) */
); /* Returns message-queue descriptor or -1 on error (sets errno) */
```

Funkcja ta zwraca deskryptor do utworzonej kolejki wiadomości.

10.1.1. Argumenty wywołania

- const char *name - identyfikator IPC POSIX
- flags - O_RDONLY, O_WRONLY, lub O_RDWR ,nie można tutaj podawać O_CREAT

Aby móc podać flagę O_CREAT należy skorzystać z innej wersji funkcji, która ma cztery argumenty:

```
mqd_t mq_open(
  const char *name, /* POSIX IPC name */
  int flags, /* flags (including O_CREAT) */
  mode_t perms, /* permissions */
  struct mq_attr *attr /* attributes (or NULL) */
); /* Returns message-queue descriptor or -1 on error (sets errno) */


struct mq_attr {
  long mq_flags; /* flags */
  long mq_maxmsg; /* max number of messages */
  long mq_msgsize; /* max message size */
  long mq_curmsgs; /* number of messages currently queued */
};
```

- Do czego służy funkcja mq_open, co zwraca ?
- Jakie atrgumenty ma funkcja mq_open ?

10.2. mq_close

To funkcja służąca do zamykania kolejki wiadomości.

```
mq_close-close message queue
#include <mqueue.h>
int mq_close(
  mqd_t mqd /* message-queue descriptor */
); /* Returns zero on success or -1 on error (sets errno) */
```

- Do czego służy funkcja mq_close, co zwraca ?

10.3. mq_unlink

To funkcja, która niszczy nazwę kolejki, ale destrukcja właściwej kolejki jest odkładana na później, aż wszystkie zapisujące do niej deskryptory zostaną usunięte.

```
mq_unlink-remove message queue
#include <mqueue.h>
int mq_unlink(
  const char *name /* POSIX IPC name */
); /* Returns 0 on success or -1 on error (sets errno) */
```

10.4. mq_send

To funkcja, która służy do wysyłania wiadomości przez kolejkę wiadomości.

```
mq_send-send message
#include <mqueue.h>
int mq_send(
  mqd_t mqd, /* message-queue descriptor */
  const char *msg, /* message */
  size_t msgsize, /*
  size of message */
  unsigned priority /*
  priority */
); /* Returns 0 on success or -1 on error (sets errno) */
```

Funkcja mq_send umieszcza wiadomość o rozmiarze msgsize w kolejce na pozycji przed wszystkimi wiadomościami z mniejszym priorytetem, ale po wiadomościach z tym samym priorytetem. Priorytety to liczby od 0 do 31.

- Do czego służy funkcja mq_send co zwraca ?
- Jakie argumenty przyjmuje funkcja mq_send ?

10.5. mq_receive

To funkcja, która odbiera najstarszą wiadomość o najwyższym priorytecie z kolejki.

```c
mq_receive-receive message
#include <mqueue.h>
ssize_t mq_receive(
  mqd_t mqd, /* message-queue descriptor */
  char *msg, /* message buffer */
  size_t msgsize, /* size of message buffer */
  unsigned *priorityp /* returned priority or NULL */
); /* Returns size of message or -1 on error (sets errno) */
```

- Do czego służy funkcja mq_receive, co zwraca ?
- Jakie argumenty przyjmuje funkcja mq_receive ?