7 Kolejki komunikatów

7.1 Wprowadzenie

Podobnie jak w przypadku semaforów i pamięci dzielonej, w nowszych wersjach systemów uniksowych (np. w Linuksie od wersji jądra 2.6.6 z biblioteką glibc od wersji 2.3.4) dostępne są kolejki komunikatów standardu POSIX, służące do komunikacji międzyprocesowej przy użyciu mechanizmu przekazywania komunikatów. Różnią się one od kolejek komunikatów Systemu V (opisanych w dodatku B) interfejsem programisty (API), a także pewnymi aspektami funkcjonalności.

Schemat użycia kolejek komunikatów standardu POSIX jest następujący: najpierw jakiś proces musi utworzyć kolejkę komunikatów, a inny proces, który chce z niej korzystać musi ją otworzyć. Następnie procesy te mogą wymieniać między sobą informację w formie komunikatów wysyłanych/odbieranych do/z takiej kolejki. Komunikaty mają przypisane priorytety i są dostarczane do odbiorcy w kolejności od najwyższego do najniższego, a dla tego samego priorytetu w kolejności FIFO. Priorytety komunikatów opisane są przy użyciu liczb całkowitych z zakresu od 0 (najniższy) do sysconf(_SC_MQ_PRIO_MAX) -1 (najwyższy)[5]. Kiedy proces skończy korzystać z kolejki komunikatów, to powinien ją zamknąć, a kiedy kolejka nie jest już potrzebna, to powinna zostać usunięta przez właściwy proces. Systemowe ograniczenia na rozmiar kolejek komunikatów standardu POSIX można uzyskać z poziomu powłoki poleceniem `ulimit -q` (wszystkie aktualne ograniczenia systemowe podawane są po wykonaniu `ulimit -a`).

W systemie Linux kolejki komunikatów standardu POSIX tworzone są w wirtualnym systemie plików i zwykle montowane pod `/dev/mqueue`.

→ Uwaga: Aby można było używać powyższych kolejek komunikatów w programach w języku C, należy je linkować z opcją: -lrt.

7.2 Tworzenie/otwieranie i usuwanie kolejki komunikatów

Do tworzenia nowej lub otwierania już istniejącej kolejki komunikatów standardu POSIX służy funkcja mq_open, przedstawiona w poniższej tabeli. Funkcja ta w przypadku

Pliki włączane	`<fcntl.h>, <sys/stat.h>, <mqueue.h>`		

Prototyp	`mqd_t mq_open(const char *name, int oflag);`
	`mqd_t mq_open(const char *name, int oflag,`
	`mode_t mode, struct mq_attr *attr);`		

Zwracana wartość	Sukces	Porażka	Czy zmienia `errno`
	deskryptor kolejki komunikatów	`(mqd_t)-1`	Tak

pomyślnego wykonania zwraca wywołującemu ją procesowi deskryptor utworzonej lub

[5]Informacje na temat funkcji sysconf można znaleźć w `man sysconf`.

otwartej kolejki komunikatów, który może być następnie używany przez inne funkcje do operacji na takiej kolejce.

• Parametry:

`name`	nazwa kolejki komunikatów zaczynająca się od znaku ukośnika (maksymalnie `NAME_MAX`, tj. 255, znaków),
`oflag`	opcje,
`mode`	prawa dostępu do obiektu (podobie jak dla pliku),
`attr`	wskaźnik do struktury atrybutów kolejki komunikatów.

• Opcje `oflag` (ważniejsze):

`O_RDONLY`	otwórz kolejkę do odbierania z niej komunikatów,
`O_WRONLY`	otwórz kolejkę do wysyłania do niej komunitatów,
`O_RDWR`	otwórz kolejkę do odbierania i wysyłania komunikatów,
`O_CREAT`	jeśli kolejka nie istnieje, to stwórz ją,
`O_EXCL`	przy równocześnie ustawionej fladze `O_CREAT` przekaż błąd, jeśli obiekt już istnieje,
`O_NONBLOCK`	otwórz kolejkę w trybie nieblokującym.

Jedną z pierwszych trzech powyższych opcji można łączyć z dowolną z trzech pozostałych przy pomocy sumy bitowej, np. `O_RDONLY | O_CREAT | O_EXCL`. Jeżeli zostanie użyta opcja `O_CREAT`, to konieczne jest dostarczenia dwóch dodatkowych argumentów: mode i attr, z których ostatni jest wskaźnikiem na strukturę atrybutów kolejki postaci:

```
struct mq_attr {
     long mq_flags;    /* Opcje: 0 lub O_NONBLOCK */
     long mq_maxmsg;   /* Maksymalna liczba komunikatow w kolejce */
     long mq_msgsize;  /* Maksymalny rozmiar komunikatu (w bajtach) */
     long mq_curmsgs;  /* Liczba komunikatow aktualnie w kolejce */
};
```

Dla funkcji mq_open pierwsza i ostatnia składowa powyższej struktury są ignorowane, można ustawiać jedynie dwie środkowe. Jeśli wskaźnik `attr` ustawiony jest na `NULL`, to tworzona jest kolejka z domyślnymi ustawieniami dla danej implementacji.

W przypadku użycia opcji `O_NONBLOCK`, kolejka zostanie otwarta w trybie *nieblokującym*, tzn. odpowiednia funkcja wysyłająca komunikat do pełnej kolejki lub odbierająca komunikat z pustej kolejki zakończy się błędem, a zmienna errno zostanie ustawiona na wartość `EAGAIN`. Bez tej opcji kolejka otwierana jest w trybie *blokującym*, tzn. powyższe funkcje odpowiednio się zablokują bez zgłaszania błędu, co oznacza, że proces odbierający komunikat będzie musiał poczekać aż taki komunikat pojawi się w kolejce, a proces wysyłający komunikat będzie czekał aż zwolni się wystarczająca ilość miejsca w kolejce.

Kiedy kolejka komunikatów nie jest już potrzebna w procesie, to można ją zamknąć przy pomocy funkcji mq_close, przedstawionej w poniższej tabeli. Zamyka ona deskryptor mqdes zwrócony wcześniej przez funkcję mq_open.

Kolejkę komunikatów można usunąć używając funkcji mq_unlink z argumentem będącym nazwą tej kolejki. Kolejki komunikatów standardu POSIX, podobnie jak semafory

Pliki włączane	`<mqueue.h>`		
Prototyp	`int mq_close(mqd_t mqdes);`		
Zwracana wartość	Sukces	Porażka	Czy zmienia `errno`
	0	−1	Tak

i pamięć dzielona, są obiektami trwałymi jądra (ang. *kernel persistence*), więc jeśli taka kolejka nie zostanie usunięta funkcją mq_unlink, to będzie istnieć aż do ponownego uruchomienia systemu.

Pliki włączane	`<mqueue.h>`		
Prototyp	`int mq_unlink(const char *name);`		
Zwracana wartość	Sukces	Porażka	Czy zmienia `errno`
	0	−1	Tak

7.3 Pobieranie/ustawianie atrybutów kolejki komunikatów

Funkcje mq_getattr i set_getattr, przedstawione w poniższej tabeli, służą odpowiednio do uzyskiwania i ustawiania atrybutów kolejki komunikatów o deskryptorze mqdes. Pierwsza

Pliki włączane	`<mqueue.h>`		

Prototypy	`int mq_getattr(mqd_t mqdes, struct mq_attr *attr);`
	`int mq_setattr(mqd_t mqdes, struct mq_attr *newattr,`
	`struct mq_attr *oldattr);`		

Zwracana wartość	Sukces	Porażka	Czy zmienia `errno`
	0	−1	Tak

z nich zwraca aktualne atrybuty kolejki przy pomocy wskaźnika attr do struktury struct mq_attr, opisanej w poprzednim podrozdziale. Natomiast druga z tych funkcji ustawia atrybuty kolejki przy pomocy wskaźnika newattr na odpowiedni obiekt struktury struct mq_attr. Jedyną modyfikacją, którą może ona wykonać na atrybutach kolejki jest ustawienie mq_flags na `O_NONBLOCK`. Jeżeli wskaźnik oldattr nie jest ustawiony na `NULL`, to struktura, na którą on wskazuje zawiera tę samą informację, jaką zwraca funkcja mq_getattr.

7.4 Wysyłanie i odbieranie komunikatów

Do wysyłania komunikatów do kolejki służy funkcja mq_send, przestawiona w poniższej tabeli. Dodaje ona komunikat o długości msg_len wskazywany przez msg_ptr do kolejki o deskryptorze mqdes. Wartość msg_len nie może przekraczać wartości atrybutu kolejki mq_msgsize w strukturze struct mq_attr (komunikaty zerowej długości są dopuszczalne). Argument msg_prio określa priorytet komunikatu, opisany w podrozdziale 7.1. Komunikaty są umieszczane w kolejce w porządku malejącego priorytetu, a dla tego samego

Pliki włączane	`<mqueue.h>`		
Prototypy	`int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned int msg_prio);`		
Zwracana wartość	Sukces	Porażka	Czy zmienia `errno`
	0	−1	Tak

priorytetu w kolejności FIFO. Domyślnie funkcja mq_send blokuje się, kiedy w kolejce nie na wystarczająco dużo miejsca na umieszczenie komunikatu. Jeżeli ustawiona jest opcja `O_NONBLOCK`, to funkcja mq_send w takim przypadku kończy się natychmiast i zgłasza błąd ustawiając zmienną errno na kod `EAGAIN`.

Funkcja mq_receive, przestawiona w poniższej tabeli, pobiera i usuwa z kolejki o deskryptorze mqdes najstarszy komunikat o nawyższym priorytecie. Argument msg_ptr

Pliki włączane	`<mqueue.h>`		
Prototypy	`int mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int *msg_prio);`		
Zwracana wartość	Sukces	Porażka	Czy zmienia `errno`
	0	−1	Tak

wskazuje na bufor, w którym komunikat zostanie umieszczony, a msg_len określa rozmiar tego bufora – powinien być nie mniejszy od atrybutu kolejki mq_msgsize w strukturze struct mq_attr. Jeżeli argument msg_prio jest różny od `NULL`, to pod wskazany adres zwracany jest priorytet komunikatu. W przypadku gdy kolejka jest pusta, funkcja mq_receive domyślnie blokuje się do czasu pojawienia się komunikatu, chyba że została ustawiona opcja `O_NONBLOCK` – wówczas kończy się natychmiast i zgłasza błąd ustawiając zmienną errno na kod `EAGAIN`.

Ponadto dostępne są funkcje mq_timedsend i mq_timedreceive, w których można określić jak długo operacje wysyłania i odbierania komunikatów mają się blokować na odpowiednio pełnej i pustej kolejce. Istnieje też funkcja mq_notify umożliwiająca procesowi rejestrację asynchronicznego powiadomienia o nowym komunikacie pojawiającym się w pustej kolejce. Więcej szczegółów na temat tych funkcji można znaleźć w man mq_overview.

ĆWICZENIE 8: Klient–Serwer: Kolejki Komunikatów

Proces *klienta* wysyła do procesu *serwera* żądanie wykonania działania arytmetycznego na dwóch liczbach postaci: liczba $\odot$ liczba, gdzie operator $\odot \in \{+, -, *, /\}$, np. 2 + 3. Serwer wykonuje odpowiednie działanie i odsyła wynik do klienta. Klient odbiera ten wynik i wypisuje go na ekranie. Posługując się mechanizmem **kolejek komunikatów** standardu POSIX zaimplementować powyższe zadanie typu **klient–serwer** z możliwością obsługi przez serwera *wielu klientów* naraz.

Niech serwer utworzy kolejkę komunikatów w trybie do odbierania o nazwie zdefiniowanej we wspólnym pliku nagłówkowym włączanym w plikach źródłowych programów serwera i klienta. Do tej kolejki klient będzie wysyłał swoje komunikaty, a serwer będzie je z niej odbierał. Klient z kolei niech utworzy kolejkę komunikatów w trybie do odbierania o nazwie `/PID`, gdzie `PID` jest jego identyfikatorem procesu PID, np. `/17895` (do utworzenia tej nazwy użyć np. funkcji sprintf). Następnie niech klient otworzy kolejkę serwera w trybie do nadawania komunikatów i w pętli wczytuje z klawiatury żądane działanie (np. używając funkcji fgets), tworzy komunikat umieszczając na początku swój PID, a po nim wczytane wyrażenie (np. przy pomocy funkcji sprintf), po czym wysyła taki komunikat do kolejki serwera. Pętlę można zakończyć znakiem końca pliku `EOF` (z klawiatury wysyła się go sekwencją klawiszy [Ctrl D]), po czym klient powinien zamknąć i usunąć własną kolejkę oraz zamknąć kolejkę serwera – czynności te umieścić w funkcji rejestrowanej przez atexit oraz w obsłudze sygnału `SIGINT`.

Serwer niech działa w pętli nieskończonej (proces *demon*), próbując odbierać komunikaty ze swojej kolejki. Po otrzymaniu komunikatu od klienta, serwer powinien odczytać z niego PID klienta (np. funkcją atoi) i otworzyć kolejkę klienta w trybie do nadawania komunikatów o nazwie postaci `/PID` (użyć np. funkcji sprintf). Następnie z komunikatu powinien odczytać odpowiednie działanie (można użyć np. funkcji sscanf), wykonać je i odesłać wynik w komunikacie (użyć np. funkcji sprintf) do kolejki klienta, po czym zamknąć tę kolejkę. Proces serwera można zakończyć np. sygnałem `SIGINT` (z klawiatury sekwencją klawiszy [Ctrl C]), z tym że serwer powinien przechwycić ten sygnał i wykonać jego obsługę w postaci zamknięcia i usunięcia własnej kolejki komunikatów – czynności te umieścić w funkcji rejestrowanej przez atexit.

Niech procesy serwera i klienta wypisują na ekranie odpowiednie komunikaty, w szczególności atrybuty kolejek komunikatów zaraz po ich utworzeniu. Uruchamiać *każdy* proces z *innego* terminala, np. użyć polecenia:

```
xterm -hold -title SERWER -bg red -e ./serwer.x &
xterm -hold -title KLIENT1 -bg green -e ./klient.x &
xterm -hold -title KLIENT2 -bg green -e ./klient.x &
...
```

w pliku `Makefile` do uruchomienia serwera i kilku klientów (więcej szczegółów można znaleźć w `man xterm`).

Podobnie jak dla semaforów i pamięci dzielonej, stworzyć własną bibliotekę funkcji do obsługi kolejek komunikatów.