WOJSKOWA AKADEMIA TECHNICZNA

im. Jarosława Dąbrowskiego

**Wydział Cybernetyki**



**SPRAWOZDANIE**

**PROJEKT INDYWIDUALNY**

**Przedmiot:** *Systemy Operacyjne*

**Autor:** *Hubert Gołaszewski*

**Grupa:** *K4X2S1*

**Data wykonania sprawozdania:** *01.01.2016r.*

# 1. OPIS ZADANIA

Opracować zestaw programów typu *producent - konsument* realizujących następujący schemat synchronicznej komunikacji międzyprocesowej:

* *proces 1*: czyta dane ze standardowego strumienia wejściowego i przekazuje je w niezmienionej formie do *procesu 2* poprzez mechanizm komunikacyjny **K1***.*
* *proces 2*: pobiera dane przesłane przez *proces 1*. Oblicza liczbę znaków w każdej linii i wyznaczoną liczbę przekazuje do *procesu 3* poprzez mechanizm komunikacyjny **K2**.
* *proces 3*: pobiera dane wyprodukowane przez *proces 2* i wypisuje je na standardowym strumieniu diagnostycznym. Jednostki danych powinny zostać wyprowadzone po maksymalnie 15 w pojedynczym wierszu i oddzielone spacjami.

Wszystkie trzy procesy powinny być powoływane automatycznie z jednego procesu inicjującego. Po powołaniu procesów potomnych proces inicjujący wstrzymuje pracę. Proces inicjujący wznawia pracę w momencie kończenia pracy programu (o czym niżej), jego zadaniem jest „posprzątać” po programie przed zakończeniem działania.

Ponadto należy zaimplementować mechanizm asynchronicznego przekazywania informacji pomiędzy operatorem a procesami oraz pomiędzy procesami. Wykorzystać do tego dostępny mechanizm sygnałów.

Operator może wysłać do dowolnego procesu sygnał zakończenia działania (S1), sygnał wstrzymania działania (S2) i sygnał wznowienia działania (S3). Sygnał S2 powoduje wstrzymanie synchronicznej wymiany danych pomiędzy procesami. Sygnał S3 powoduje wznowienie tej wymiany. Sygnał S1 powoduje zakończenie działania oraz zwolnienie wszelkich wykorzystywanych przez procesy zasobów (zasoby zwalnia proces macierzysty).

Każdy z sygnałów przekazywany jest przez operatora tylko do jednego, dowolnego procesu. O tym, do którego procesu wysłać sygnał, decyduje operator, a nie programista. Każdy z sygnałów operator może wysłać do innego procesu. Mimo, że operator kieruje sygnał do jednego procesu, to pożądane przez operatora działanie musi zostać zrealizowane przez wszystkie trzy procesy. W związku z tym, proces odbierający sygnał od operatora musi powiadomić o przyjętym żądaniu pozostałe dwa procesy. Powinien wobec tego przekazać do nich odpowiedni sygnał informując o tym jakiego działania wymaga operator. Procesy odbierające sygnał, powinny zachować się adekwatnie do otrzymanego sygnału. Wszystkie trzy procesy powinny zareagować zgodnie z żądaniem operatora.

Sygnały oznaczone w opisie zadania symbolami S1, S2, S3 należy wybrać samodzielnie spośród dostępnych w systemie. Program ma umożliwiać uruchomienie:

* w trybie interaktywnym – operator wprowadza dane z klawiatury,
* w trybie odczytu danych z określonego pliku,
* w trybie odczytu danych z pliku /dev/urandom.

**2. KOD ŹRÓDŁOWY**

/\*###########################################################################################################################

#WOJSKOWA AKADEMIA TECHNICZNA

#PROJEKT: ZESTAW PROGRAMOW TYPU PRODUCENT - KONSUMENT REALIZUJACYCH SYNCHRONICZNA KOMUNIKACJE MIEDZYPROCESOWA

#PRZEDMIOT: SYSTEMY OPERACYJNE

#RODZAJ ZAJEC: LABORATORIUM

#GRUPA: K4X2S1

#AUTOR: HUBERT GOŁASZEWSKI

#ROK: 2015

@MECHANIZM K1: SHARED MEMORY

@MECHANIZM K2: PIPE

@SYGNAL ZAKONCZENIA: SIGQUIT (3)

@SYGNAL ZATRZYMANIA: SIGILL (4)

@SYGNAL WZNOWIENIA: SIGTRAP (5)

#############################################################################################################################\*/

#include <semaphore.h> // sem\_open(), sem\_wait(), sem\_post(), sem\_t TYPE

#include <stdlib.h> // exit()

#include <stdio.h> // printf()

#include <unistd.h> // fork(), getpid(), sleep(), read(), etc.

#include <fcntl.h> // sem\_open(O\_CREAT)

#include <stdbool.h> // bool TYPE

#include <sys/shm.h> // shmget(), shmat(), shmdt(), shmctl(), etc.

#include <sys/ipc.h> // key\_t TYPE

#include <sys/types.h> // key\_t TYPE, pid\_t TYPE

#include <signal.h> // signal()

#include <time.h> // usleep()

#include <errno.h> // perror()

#define CHARLIMIT 1 // Wczytywanie linii po jednym znaku

#define FILENAMELIMIT 60 // Nazwa wlasnego pliku nie bedzie przekraczac 60 znakow

#define RDX 0 // Deskryptor odczytu dla lacza nienazwanego

#define WRX 1 // Deskryptor zapisu dla lacza nienazwanego

char semNameProduction[] = "/1SEMAPHOREproductionPOSIXn"; // Nazwanie semafora producenta

char semNameConsumption[] = "/2SEMAPHOREconsumptionPOSIXn"; // Nazwanie semafora konsumenta pierwszego

char semNameConsumption2[] = "/3SEMPAHOREconsumption2POSIXn"; // Nazwanie semafora konsumenta drugiego

char semNameEnd[] = "/4SEMAPHOREendPOSIXn"; // Nazwanie semafora konca

sem\_t \*mutexProduction; // Semefor produkcyjny obslugiwany przez P1/P2/P3

sem\_t \*mutexConsumption; // Semafor konsumpcyjny pierwszy obslugiwany przez P1/P2

sem\_t \*mutexConsumption2; // Semefor konsumpcyjny drugi obslugiwany przez P2/P3

sem\_t \*mutexEnd; // Semafor konca obslugiwany przez PM/P3

int fileData; // Deskryptor pliku

char fileName[FILENAMELIMIT] = "/dev/urandom"; // Tablica przechowujaca nazwe pliku

int dsxALPHA[2]; // Uchwyty na deskryptory odczytu/zapisu dla pipe

pid\_t PID1; // PID procesu P1

pid\_t PID2; // PID procesu P2

pid\_t PID3; // PID procesu P3

char \*flagsPointer; // Wskaznik na pamiec dzielona przechowujaca flagi stanu i flage trybu

int flagsSegID; // ID segmentu pamieci dzielonej flagowej

char \*memoryPointer; // Wskaznik na pamiec dzielona przechowujaca pojedyncze znaki z ciagu

int charSegID; // ID segmentu pamieci dzielonej znakowej

void quitP1(int); // Prototyp funkcji quitP1()

void quitP2(int); // Prototyp funkcji quitP2()

void quitP3(int); // Prototyp funkcji quitP3()

void stopP1(int); // Prototyp funkcji stopP1()

void stopP2(int); // Prototyp funkcji stopP2()

void stopP3(int); // Prototyp funkcji stopP3()

void resumeP1(int); // Prototyp funkcji resumeP1()

void resumeP2(int); // Prototyp funkcji resumeP2()

void resumeP3(int); // Prototyp funkcji resumeP3()

void systemCleanPM(void); // Prototyp funkcji systemCleanPM()

int clearBuffer(char); // Prototyp funkcji clearBuffer()

int main(int argc, const char \*argv[]){

setbuf(stdout, NULL); // Wylaczenie buforowania STDOUT

key\_t KEY\_flags = 0x2AB; // Identyfikator segmentu pamieci flagowej

if((flagsSegID = shmget(KEY\_flags, sizeof(char)\*4, 0666 | IPC\_CREAT)) < 0) perror("SHMGET(FLAGS) - PM"); // Pobranie ID pamieci wspoldzielonej

if((flagsPointer = shmat(flagsSegID, NULL, 0)) == (char \*) -1) perror("SHMAT(FLAGS) - PM"); // Przylaczenie sie do segmentu pamieci dzielonej flagowej

\*flagsPointer = 'W'; // Ustawienie flagi procesu P1 na 'W' - pracujacy

\*(flagsPointer+1) = 'W'; // Ustawienie flagi procesu P2 na 'W' - pracujacy

\*(flagsPointer+2) = 'W'; // Ustawienie flagi procesu P3 na 'W' - pracujacy

char data[CHARLIMIT] = {}; // Tablica przechowujaca pojedyncze znaki z ciagu

key\_t KEY\_char = 0x29A; // Identyfikator segmentu pamieci znakowej

if((charSegID = shmget(KEY\_char, sizeof(data), 0666 | IPC\_CREAT)) < 0) perror("SHMGET(CHAR) - PM"); // Pobranie ID segmentu pamieci wspoldzielonej

if((memoryPointer = shmat(charSegID, NULL, 0)) == (char \*) -1) perror("SHMAT(CHAR) - PM"); // Przylaczenie sie do segmentu pamieci dzielonej znakowej

int signalIgnore = 1; // Zmienna przechowujaca numer sygnalu do zignorowania

for(signalIgnore; signalIgnore <= 64; signalIgnore++){

signal(signalIgnore, SIG\_IGN); // Zignorowanie wszystkich sygnalow przez proces PM

}

if((mutexProduction = sem\_open(semNameProduction, O\_CREAT, 0666, 1)) == SEM\_FAILED) perror("SEM\_OPEN(P) - PM"); // Stworzenie semafora producenta i podniesienie go

if((mutexConsumption = sem\_open(semNameConsumption, O\_CREAT, 0666, 0)) == SEM\_FAILED) perror("SEM\_OPEN(C1) - PM"); // Stworzenie semafora konsumenta pierwszego i opuszczenie go

if((mutexConsumption2 = sem\_open(semNameConsumption2, O\_CREAT, 0666, 0)) == SEM\_FAILED) perror("SEM\_OPEN(C2) - PM"); // Stworzenie semafora konsumenta drugiego i opuszczenie go

if((mutexEnd = sem\_open(semNameEnd, O\_CREAT, 0666, 0)) == SEM\_FAILED) perror("SEM\_OPEN(E) - PM"); // Stworzenie semafora konca i podniesienie go

if(!fork()){ // STWORZENIE PROCESU PIERWSZEGO

if((sem\_close(mutexConsumption2)) == -1) perror("SEM\_CLOSE(C2) - P1"); // Zamkniecie semafora konsumenta drugiego dla procesu P1

if((sem\_close(mutexEnd)) == -1) perror("SEM\_CLOSE(E) - P1"); // Zamkniecie semafora konca dla procesu P1

PID1 = getpid(); // Pobranie PIDu procesu P1

signal(SIGILL, stopP1); // Obsluga sygnalu zatrzymania

signal(SIGQUIT, quitP1); // Obsluga sygnalu zakoczenia

char option; // Opcja wprowadzania danych

int decision = -1; // Zmienna okreslajaca czy wybrano prawidlowa opcje

do{ // Menu

printf("CHOOSE YOUR INPUT METHOD: \n");

printf("1) KEYBOARD\n");

printf("2) YOUR FILE\n");

printf("3) /DEV/URANDOM FILE\n");

printf("OPTION: ");

option = getchar(); // Podaj opcje wprowadzania danych i ignoruj znak '\n' poniewaz zostalby w buforze

decision = clearBuffer(option); // Wyczyszczenie bufora i sprawdzenie poprawnosci wybranej opcji

fflush(stdout); // Wyczyszczenie bufora dla pewnosci (prawdopodobnie nie dziala na Linuxie)

fflush(stdin); // Wyczyszczenie bufora dla pewnosci (prawdopodobnie nie dziala na Linuxie)

printf("\033[H\033[J"); // Wyczyszczenie ekranu

}

while(decision != 1); // Pytaj dopoki decyzcja nie bedzie pozytywna

if(option == '1'){ // Jezeli wybrano wprowadzanie z klawiatury

bool end = false; // Zmienna okreslajaca czy pobrano caly ciag (wypisywanie nowych linii)

bool first = true; // Zmienna pomagajaca wyswietlic prawidlowo wyniki dla option == '1'

\*(flagsPointer+3) = 'K'; // Podniesienie flagi trybu 'K' - keyboard

while(true){

sem\_wait(mutexProduction); // Opuszczenie semafora producenta

if(first == false && end == true) printf("\n"); // Poprawne wyswietlenie wynikow

if(end == true) end = false; // Stwierdzenie ze wczytywany jest nowy ciag

read(0, &data, CHARLIMIT); // Wczytywanie danych znak po znaku do tablicy 1 - elementowej z STDIN

\*memoryPointer = data[0]; // Zapisanie znaku do pamieci wspoldzielonej

if(data[0] == '\n') end = true; // Jezeli wystapil znak nowej linii to zmienna end = true

if(first == true) first = false; // Stwierdzenie, ze od tej pory nie jest to pierwsze przejscie petli

sem\_post(mutexConsumption); // Podniesienie semafora konsumenta pierwszego

}

}

else if(option == '2'){ // Jezeli wybrano wprowadzanie z wlasnego pliku

\*(flagsPointer+3) = 'O'; // Podniesienie flagi trybu 'O' - own file

do{

printf("INPUT YOUR FILE NAME: ");

scanf("%s", fileName); // Podanie nazwy pliku z danymi

fileData = open(fileName, O\_RDONLY); // Otwarcie pliku tylko do odczytu

printf("\033[H\033[J"); // Wyczyszczenie ekranu

if(fileData == -1) perror("OPEN() - OWN FILE"); // Jezeli zla nazwa zwroc blad

}

while(fileData == -1); // Wczytywanie nazwy dopoki bedzie prawidlowa

while(true){

sem\_wait(mutexProduction); // Opuszczenie semafora producenta

if(read(fileData, &data, CHARLIMIT) == 0) quitP1(3); // Wczytywanie danych do tablicy 1 - elementowej, jezeli EOF zakoncz program

\*memoryPointer = data[0]; // Zapisanie znaku do pamieci dzielonej

sem\_post(mutexConsumption); // Podniesienie semafora konsumenta pierwszego

}

}

else if(option == '3'){ // Jezeli wybrano wprowadzanie z pliku /dev/urandom

\*(flagsPointer+3) = 'U'; // Podniesienie flagi trybu 'U' - urandom file

do{

fileData = open(fileName, O\_RDONLY); // Otwarcie pliku tylko do odczytu

printf("\033[H\033[J"); // Wyczyszczenie ekranu

if(fileData == -1){

perror("OPEN() - URANDOM FILE"); // Jezeli zla sciezka zwroc blad

printf("INPUT URANDOM FILE PATH: ");

scanf("%s", fileName); // Podanie nazwy pliku z danymi

}

}

while(fileData == -1); // Wczytywanie nazwy dopoki bedzie prawidlowa

while(true){

sem\_wait(mutexProduction); // Opuszczenie semafora producenta, $

read(fileData, &data, CHARLIMIT); // Wczytywanie danych do tablicy 1 - elementowej

\*memoryPointer = data[0]; // Zapisanie znaku do pamieci dzielonej

sem\_post(mutexConsumption); // Podniesienie semafora konsumenta pierwszego

}

}

exit(1); // Niepoprawne zamkniecie procesu

}

if((pipe(dsxALPHA)) == -1) perror("PIPE() - PM"); // Utworzenie lacza nienazwanego

if(!fork()){ // STWORZENIE PROCESU DRUGIEGO

if((close(dsxALPHA[RDX])) == -1) perror("CLOSE(RDX) - P2"); // Zamkniecie odczytu z potoku dla procesu P2 (jako producent)

if((sem\_close(mutexEnd)) == -1) perror("SEM\_CLOSE(E) - P2"); // Zamkniecie semafora konca dla procesu P2

PID2 = getpid(); // Pobranie PIDu procesu P2

signal(SIGILL, stopP2); // Obsluga sygnalu zatrzymania

signal(SIGQUIT, quitP2); // Obsluga sygnalu zakonczenia

char data[CHARLIMIT] = {}; // Tablica przechowujaca dane znakowe, zainicjalizowana na 0

int counter = 0; // Licznik znakow

bool end = false; // Zmienna

while(true){ // Odczytywanie danych z pamieci dzielonej od procesu P1 i wysylanie laczem do procesu P3

sem\_wait(mutexConsumption); // Opuszczenie semafora konsumenta pierwszego

if(end == true) end = false; // Stwierdzenie ze wczytywany jest nowy ciag

counter++; // Zliczanie znakow w pojedynczym ciagu

if(\*memoryPointer != '\n') sem\_post(mutexProduction); // Jesli nie pobrano calego ciagu, wczytuj dalej

else{ // Jesli pobrano caly ciag przekaz liczbe znakow do procesu P3

end = true; // Stwierdzenie ze wczytano caly ciag

write(dsxALPHA[WRX], &counter, sizeof(counter)); // Przekazanie liczby znakow do procesu P3

counter = 0; // Wyzerowanie licznika znakow

sem\_post(mutexConsumption2); // Podniesienie semafora konsumenta drugiego

}

}

exit(1); // Niepoprawne zamkniecie procesu

}

if(!fork()){ // STWORZENIE PROCESU TRZECIEGO

if((sem\_close(mutexConsumption)) == -1) perror("SEM\_CLOSE(C1) - P3"); // Zamkniecie semafora konsumenta pierwszego dla procesu P3

if((shmdt(memoryPointer)) == -1) perror("SHMDT(CHAR) - P3"); // Odlaczenie pamieci dzielonej znakowej dla procesu P3

if((close(dsxALPHA[WRX])) == -1) perror("CLOSE(WRX) - P3"); // Zamkniecie zapisu do potoku dla procesu P3

PID3 = getpid(); // Pobranie PIDu procesu P3

signal(SIGILL, stopP3); // Obsluga sygnalu zatrzymania

signal(SIGQUIT, quitP3); // Obsluga sygnalu zakonczenia

int value; // Ilosc znakow we wczytanej linii

int newline = 0; // Zmienna okreslajaca ile jednostek wynikow ma byc wypisywanych w jednej linii

while(\*(flagsPointer+2) != 'Q'){

sem\_wait(mutexConsumption2); // Opuszczenie semafora konsumenta drugiego

read(dsxALPHA[RDX], &value, sizeof(value)); // Odczytanie liczby znakow od procesu P2

printf("%d ", value); // Wypisanie liczby znakow na STDOUT

if(\*(flagsPointer+3) != 'K'){

newline++; // Zwiekszenie licznika jednostek

if(newline == 10){ // Jezeli wystapilo 10 jednostek to wypisz znak nowej linii i zresetuj licznik jednostek

printf("\n");

newline = 1;

}

}

sem\_post(mutexProduction); // Podniesienie semafora producenta

}

exit(1); // Niepoprawne zamkniecie procesu

}

if((close(dsxALPHA[RDX])) == -1) perror("CLOSE(RDX) - PM"); // Zamkniecie odczytu do potoku dla procesu PM

if((close(dsxALPHA[WRX])) == -1) perror("CLOSE(WRX) - PM"); // Zamkniecie zapisu do potoku dla procesu PM

if((sem\_close(mutexProduction)) == -1) perror("SEM\_CLOSE(P) - PM"); // Zamkniecie semafora produkcyjnego dla procesu PM

if((sem\_close(mutexConsumption)) == -1) perror("SEM\_CLOSE(C1) - PM"); // Zamkniecie semafora konsumpcyjnego pierwszego dla procesu PM

if((sem\_close(mutexConsumption2)) == -1) perror("SEM\_CLOSE(C2) - PM"); // Zamkniecie semafora konsumpcyjnego drugiego dla procesu PM

if((shmdt(flagsPointer)) == -1) perror("SHMDT(FLAGS) - PM"); // Odlaczenie pamieci dzielonej flagowej dla procesu PM

if((shmdt(memoryPointer)) == -1) perror("SHMDT(CHAR) - PM"); // Odlaczenie pamieci dzielonej znakowej dla procesu PM

sem\_wait(mutexEnd); // Oczekiwanie na podniesienie semafora konca przez proces P3 (zakonczenie dzialania calego programu)

systemCleanPM(); // Funkcja zwalniajaca wszystkie uzyte zasoby

if((sem\_close(mutexEnd)) == -1) perror("SEM\_CLOSE(E) - PM"); // Zamkniecie semafora konca dla PM

printf("\nEND OF ALL PROCESSES!\n"); // Komunikat inforujacy o poprawnym zakonczeniu calego programu!

return 0;

}

void quitP1(int signalX){ // Funkcja konczaca proces P1

signal(SIGQUIT, SIG\_IGN); // Zignorowanie sygnalu SIGQUIT

signal(SIGILL, SIG\_IGN); // Zignorowanie sygnalu SIGILL

signal(SIGTRAP, SIG\_IGN); // Zignorowanie sygnalu SIGTRAP

if(\*flagsPointer != 'Q') \*flagsPointer = 'Q'; // Podniesienie flagi zakonczenia dla procesu P1

usleep(100000); // Zapobiega powielaniu sygnalow

if(\*(flagsPointer+1) != 'Q') kill(PID2, SIGQUIT); // Wyslanie sygnalu do procesu P2 jesli nie byl wczesniej wyslany

if(\*(flagsPointer+2) != 'Q') kill(PID3, SIGQUIT); // Wyslanie sygnalu do procesu P3 jesli nie byl wczesniej wyslany

if(\*(flagsPointer+3) != 'K') if((close(fileData)) == -1) perror("CLOSE(OWN FILE) - P1"); // Zamkniecie pliku

if((sem\_close(mutexProduction)) == -1) perror("SEM\_CLOSE(P) - P1"); // Zamkniecie semafora producenta dla procesu P1

if((sem\_close(mutexConsumption)) == -1) perror("SEM\_CLOSE(C1) - P1"); // Zamkniecie semafora konsumenta pierwszego dla procesu P1

if((shmdt(memoryPointer)) == -1) perror("SHMDT(CHAR) - P1"); // Odlaczenie pamieci dzielonej dla procesu P1

if((shmdt(flagsPointer)) == -1) perror("SHMDT(FLAGS) - P1"); // Odlaczenie pamieci dzielonej dla procesu P1

exit(0);

}

void quitP2(int signalX){ // Funkcja konczaca proces P2

signal(SIGQUIT, SIG\_IGN); // Zignorowanie sygnalu SIGQUIT

signal(SIGILL, SIG\_IGN); // Zignorowanie sygnalu SIGILL

signal(SIGTRAP, SIG\_IGN); // Zignorowanie sygnalu SIGTRAP

if(\*(flagsPointer+1) != 'Q') \*(flagsPointer+1) = 'Q'; // Podniesienie flagi zakonczenia dla procesu P2

usleep(100000); // Zapobiega powielaniu sygnalow

if(\*flagsPointer != 'Q') kill(PID1, SIGQUIT); // Wyslanie sygnalu do procesu P1 jesli nie byl wczesniej wyslany

if(\*(flagsPointer+2) != 'Q') kill(PID3, SIGQUIT); // Wyslanie sygnalu do procesu P3 jesli nie byl wczesniej wyslany

if((close(dsxALPHA[WRX])) == -1) perror("CLOSE(WRX) - P2"); // Zamkniecie zapisu do potoku dla procesu P2

if((sem\_close(mutexProduction)) == -1) perror("SEM\_CLOSE(P) - P2"); // Zamkniecie semafora konsumenta pierwszego dla procesu P2

if((sem\_close(mutexConsumption)) == -1) perror("SEM\_CLOSE(C1) - P2"); // Zamkniecie semafora konsumenta pierwszego dla procesu P2

if((sem\_close(mutexConsumption2)) == -1) perror("SEM\_CLOSE(C2) - P2"); // Zamkniecie semafora konsumenta drugiego dla procesu P2

if((shmdt(memoryPointer)) == -1) perror("SHMDT(CHAR) - P2"); // Odlaczenie pamieci dzielonej znakowej dla procesu P2

if((shmdt(flagsPointer)) == -1) perror("SHMDT(FLAGS) - P2"); // Odlaczenie pamieci dzielonej flagowej dla procesu P2

exit(0);

}

void quitP3(int signalX){ // Funkcja konczaca proces P3

signal(SIGQUIT, SIG\_IGN); // Zignorowanie sygnalu SIGQUIT

signal(SIGILL, SIG\_IGN); // Zignorowanie sygnalu SIGILL

signal(SIGTRAP, SIG\_IGN); // Zignorowanie sygnalu SIGTRAP

if(\*(flagsPointer+2) != 'Q') \*(flagsPointer+2) = 'Q'; // Podniesienie flagi zakonczenia dla procesu P3

usleep(100000); // Zapobiega powielaniu sygnalow

if(\*flagsPointer != 'Q') kill(PID1, SIGQUIT); // Wyslanie sygnalu do procesu P1 jesli nie byl wczesniej wyslany

if(\*(flagsPointer+1) != 'Q') kill(PID2, SIGQUIT); // Wyslanie sygnalu do procesu P2 jesli nie byl wczesniej wyslany

if((close(dsxALPHA[RDX])) == -1) perror("CLOSE(RDX) - P3"); // Zamkniecie odczytu do potoku dla procesu P3

if((sem\_close(mutexProduction)) == -1) perror("SEM\_CLOSE(P) - P3"); // Zamkniecie semafora producenta dla procesu P3

if((sem\_close(mutexConsumption2)) == -1) perror("SEM\_CLOSE(C2) - P3"); // Zamkniecie semafora konsumenta drugiego dla procesu P3

if((shmdt(flagsPointer)) == -1) perror("SHMDT(FLAGS) - P3"); // Odlaczenie pamieci dzielonej flagowej dla procesu P3

sem\_post(mutexEnd); // Podniesienie semafora konca (oznacza zakonczenie dzialania programu)

if((sem\_close(mutexEnd)) == -1) perror("SEM\_CLOSE(E) - P3"); // Zamkniecie semafora konca dla procesu P3

exit(0);

}

void stopP1(int signalX){ // Funkcja zatrzymujaca proces P1

signal(SIGTRAP, resumeP1); // Obsluga sygnalu SIGTRAP

signal(SIGQUIT, quitP1); // Obsluga sygnalu SIGQUIT

signal(SIGILL, SIG\_IGN); // Zignorowanie sygnalu SIGILL

if (\*flagsPointer == 'W') \*flagsPointer = 'S'; // Podniesienie flagi zatrzymania dla procesu P1

usleep(100000); // Zapobiega powielaniu sygnalow

if(\*(flagsPointer+1) == 'W') kill(PID2, SIGILL); // Wyslanie sygnalu do procesu P2 jesli nie byl wczesniej wyslany

if(\*(flagsPointer+2) == 'W') kill(PID3, SIGILL); // Wyslanie sygnalu do procesu P3 jesli nie byl wczesniej wyslany

pause(); // Zatrzymanie procesu P1

signal(SIGILL, stopP1); // Przywrocenie starej obslugi sygnalu SIGILL

signal(SIGQUIT, quitP1); // Przywrocenie starej obslugi sygnalu SIGQUIT

}

void stopP2(int signalX){ // Funkcja zatrzymujaca proces P2

signal(SIGTRAP, resumeP2); // Obsluga sygnalu SIGTRAP

signal(SIGQUIT, quitP2); // Obsluga sygnalu SIGQUIT

signal(SIGILL, SIG\_IGN); // Zignorowanie sygnalu SIGILL

if (\*(flagsPointer+1) == 'W') \*(flagsPointer+1) = 'S'; // Podniesienie flagi zatrzymania dla procesu P2

usleep(100000); // Zapobiega powielaniu sygnalow

if(\*(flagsPointer) == 'W') kill(PID1, SIGILL); // Wyslanie sygnalu do procesu P1 jesli nie byl wczesniej wyslany

if(\*(flagsPointer+2) == 'W') kill(PID3, SIGILL); // Wyslanie sygnalu do procesu P3 jesli nie byl wczesniej wyslany

pause(); // Zatrzymanie procesu P2

signal(SIGILL, stopP2); // Przywrocenie starej obslugi sygnalu SIGILL

signal(SIGQUIT, quitP2); // Przywrocenie starej obslugi sygnalu SIGQUIT

}

void stopP3(int signalX){ // Funkcja zatrzymujaca proces P3

signal(SIGTRAP, resumeP3); // Obsluga sygnalu SIGTRAP

signal(SIGQUIT, quitP3); // Obsluga sygnalu SIGQUIT

signal(SIGILL, SIG\_IGN); // Zignorowanie sygnalu SIGILL

if (\*(flagsPointer+2) == 'W') \*(flagsPointer+2) = 'S'; // Podniesienie flagi zatrzymania dla procesu P3

usleep(100000); // Zapobiega powielaniu sygnalow

if(\*(flagsPointer) == 'W') kill(PID1, SIGILL); // Wyslanie sygnalu do procesu P1 jesli nie byl wczesniej wyslany

if(\*(flagsPointer+1) == 'W') kill(PID2, SIGILL); // Wyslanie sygnalu do procesu P2 jesli nie byl wczesniej wyslany

pause(); // Zatrzymanie procesu P2

signal(SIGILL, stopP3); // Przywrocenie starej obslugi sygnalu SIGILL

signal(SIGQUIT, quitP3); // Przywrocenie starej obslugi sygnalu SIGQUIT

}

void resumeP1(int signalX){ // Funkcja wznawiajaca proces P1

signal(SIGQUIT, SIG\_IGN); // Zignorowanie sygnalu SIGQUIT

signal(SIGILL, SIG\_IGN); // Zignorowanie sygnalu SIGILL

signal(SIGTRAP, SIG\_IGN); // Zignorowanie sygnalu SIGTRAP

if (\*flagsPointer == 'S') \*flagsPointer = 'W'; // Podniesienie flagi wznowienia dla procesu P1

usleep(100000); // Zapobiega powielaniu sygnalow

if(\*(flagsPointer+1) == 'S') kill(PID2, SIGTRAP); // Wyslanie sygnalu do procesu P2 jesli nie byl wczesniej wyslany

if(\*(flagsPointer+2) == 'S') kill(PID3, SIGTRAP); // Wyslanie sygnalu do procesu P3 jesli nie byl wczesniej wyslany

signal(SIGILL, stopP1); // Przywrocenie starej obslugi sygnalu SIGILL

signal(SIGQUIT, quitP1); // Przywrocenie starej obslugi sygnalu SIGQUIT

}

void resumeP2(int signalX){ // Funkcja wznawiajaca proces P2

signal(SIGQUIT, SIG\_IGN); // Zignorowanie sygnalu SIGQUIT

signal(SIGILL, SIG\_IGN); // Zignorowanie sygnalu SIGILL

signal(SIGTRAP, SIG\_IGN); // Zignorowanie sygnalu SIGTRAP

if (\*(flagsPointer+1) == 'S') \*(flagsPointer+1) = 'W'; // Podniesienie flagi wznowienia dla procesu P2

usleep(100000); // Zapobiega powielaniu sygnalow

if(\*(flagsPointer) == 'S') kill(PID1, SIGTRAP); // Wyslanie sygnalu do procesu P1 jesli nie byl wczesniej wyslany

if(\*(flagsPointer+2) == 'S') kill(PID3, SIGTRAP); // Wyslanie sygnalu do procesu P3 jesli nie byl wczesniej wyslany

signal(SIGILL, stopP2); // Przywrocenie starej obslugi sygnalu SIGILL

signal(SIGQUIT, quitP2); // Przywrocenie starej obslugi sygnalu SIGQUIT

}

void resumeP3(int signalX){ // Funkcja wznawiajaca proces P3

signal(SIGQUIT, SIG\_IGN); // Zignorowanie sygnalu SIGQUIT

signal(SIGILL, SIG\_IGN); // Zignorowanie sygnalu SIGILL

signal(SIGTRAP, SIG\_IGN); // Zignorowanie sygnalu SIGTRAP

if (\*(flagsPointer+2) == 'S') \*(flagsPointer+2) = 'W'; // Podniesienie flagi wznowienia dla procesu P3

usleep(100000); // Zapobiega powielaniu sygnalow

if(\*(flagsPointer) == 'S') kill(PID1, SIGTRAP); // Wyslanie sygnalu do procesu P1 jesli nie byl wczesniej wyslany

if(\*(flagsPointer+1) == 'S') kill(PID2, SIGTRAP); // Wyslanie sygnalu do procesu P2 jesli nie byl wczesniej wyslany

signal(SIGILL, stopP3); // Przywrocenie starej obslugi sygnalu SIGILL

signal(SIGQUIT, quitP3); // Przywrocenie starej obslugi sygnalu SIGQUIT

}

int clearBuffer(char option){ // Funkcja czyszczaca bufor po blednym wybraniu opcji z menu

char symbol; // Kolejny znak do pobrania

symbol = getchar(); // Pobierz kolejny znak

if((option == '1' || option == '2' || option == '3') && symbol == '\n') return 1; // Jezeli wybrano prawidlowo zwroc 1

while(symbol != '\n') symbol = getchar(); // Wczytywanie znaku dopoki nie napotka nowej linii

return -1; // Wybrano zle wiec zwroc -1

}

void systemCleanPM(void){ // Funkcja zwalniajaca zasoby przez PM

usleep(400000); // Danie czasu na poprawne zakonczenie procesow P1, P2 i P3

if((sem\_unlink(semNameProduction)) == -1) perror("SEM\_UNLINK() - P"); // Odlaczenie semafora producenta

if((sem\_unlink(semNameConsumption)) == -1) perror("SEM\_UNLINK() - C1"); // Odlaczenie semafora konsumenta pierwszego

if((sem\_unlink(semNameConsumption2)) == -1) perror("SEM\_UNLINK() - C2"); // Odlaczenie semafora konsumenta drugiego

if((sem\_unlink(semNameEnd)) == -1) perror("SEM\_UNLINK() - E"); // Odlaczenie semafora konca

if((shmctl(charSegID, IPC\_RMID, NULL)) == -1) perror("SHMCTL() - CHAR"); // Usuniecie pamieci dzielonej

if((shmctl(flagsSegID, IPC\_RMID, NULL)) == -1) perror("SHMCTL() - FLAGS"); // Usuniecie pamieci dzielonej

}

**3. OPIS ROZWIĄZANIA**

**Mechanizm K1:** Shared memory

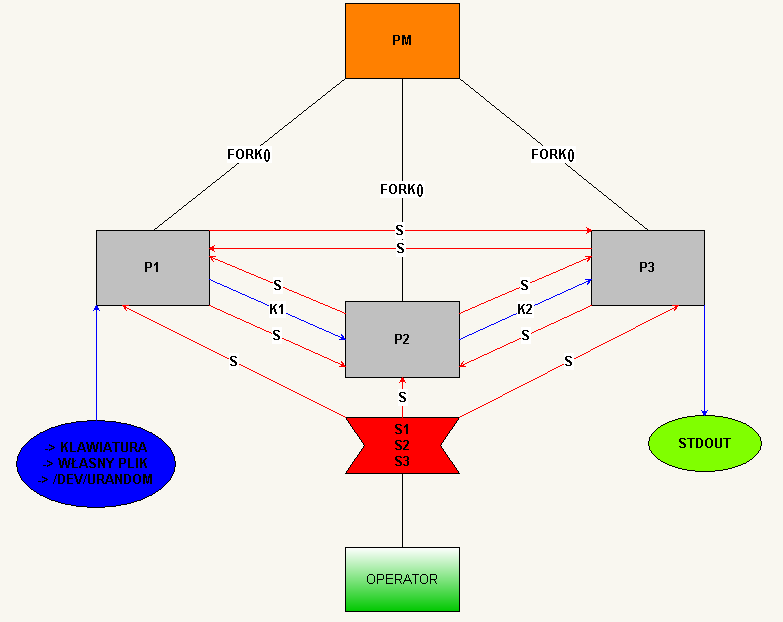
**Mechanizm K2:** Pipe

**Sygnał S1 (zakończenie):** SIGQUIT (3)

**Sygnał S2 (zatrzymanie):** SIGILL (4)

**Sygnał S3 (wznowienie):** SIGTRAP (5)

**MODEL ROZWIĄZANIA:**



**K1** - pamięć dzielona.

**K2** - łącze nienazwane (pipe).

**PM** - proces macierzysty inicjujący pozostałe trzy procesy.

**P1** - proces pierwszy pobiera dane z pliku lub STDIN i przekazuje te dane przez mechanizm K1 do procesu P2.

**P2** - proces drugi pobiera dane od procesu P1, oblicza liczbę znaków i przekazuje wynik przez mechanizm K2 do procesu P3.

**P3** - proces trzeci odbiera liczbę znaków w danej linii obliczoną przez proces P2 i wyświetla wynik na STDOUT.

**S1, S2, S3** - sygnały, które operator może wysyłać do procesów P1, P2 i P3.

Niebieskie linie obrazują wędrówkę danych.

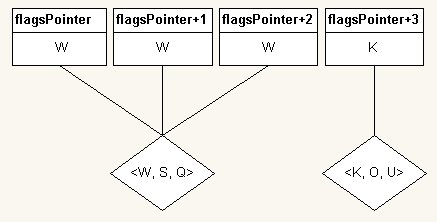
Czerwone linie między procesami obrazują możliwość wysłania/odebrania danego sygnału.

Proces macierzysty tworzy dwa obszary pamięci dzielonej i trzy procesy potomne. Jeden obszar pamięci służy do przechowywania danych czyli pojedynczego znaku z podanego ciągu, który dalej odbierany jest przez proces P2. Pamięć ta zawiera jedną komórkę typu "char".

Drugi obszar pamięci służy do przechowywania trzech flag (są trzy rodzaje) stanu procesów (flagi ustawiają sygnały) oraz flagę trybu (również dostępne są trzy rodzaje) dla całego programu. Cały ten obszar zawiera w sumie cztery komórki typu "char".

Do synchronizacji wymiany danych posłużyłem się semaforami nazwanymi POSIX. Łącznie tworzone są cztery semafory. Trzy z nich służą bezpośrednio do synchronizacji wymiany danych, a jeden służy wyłącznie to zatrzymania procesu macierzystego, aby ten nie wykonywał pustych instrukcji czekając na zakończenie. Tworzone jest również łącze nienazwane służące do komunikacji między procesem P2 i P3.

**Model pamięci dzielonej przechowującej flagi stanu procesów (przykład wariantu):**



*Na początku programu flagi stanu procesów (flagsPointer - dla P1, flagsPointer+1 dla P2, flagsPointer+2 dla P3) zostają zainicjalizowane na 'W' czyli stan pracy. Inne dostępne flagi oznaczają: S - stan stopu, Q - stan zakończenia.*

*Po wybraniu trybu pracy ostatnia flaga trybu zostaje ustawiona na jedna z trzech dostępnych: 'K' - tryb interaktywny, 'O' - tryb pliku własnego lub 'U' - tryb pliku "urandom".*

*Wszystkie semafory na starcie zostają opuszczone z wyjątkiem semafora producenta.*

Przeprowadźmy uproszczoną symulację działania programu (w dużym spowolnieniu). Na początku automatycznie zainicjalizowana zostaje pamięć dzielona stanu procesów na 'W' - 'W' - 'W' (flaga 'W' oznacza stan pracy). Załóżmy, że wybraliśmy tryb interaktywny, więc zostaje również ustawiona flaga trybu na 'K' (jest to ostatnia flaga).

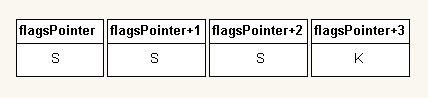
Proces P1 od razu opuszcza swój semafor i oczekuje na wprowadzenie danych (pozostałe procesy czekają na semaforach). Załóżmy, że wprowadziliśmy z klawiatury ciąg znaków "ba". Chcemy, aby proces P2 policzył ilość znaków występujących w tym ciągu (łącznie ze znakami białymi) i przekazał tę liczbę do procesu P3, który wypisze wynik na STDOUT. Proces P1 umieszcza w obszarze pamięci dzielonej pierwszą literę tego ciągu i podnosi semafor pierwszego konsumenta (dla procesu P2). Pamięć po wypełnieniu wygląda tak:



Proces P2 opuszcza swój semafor, inkrementuje zmienną przechowująca ilość znaków w ciągu, po czym stwierdza, iż odebranym znakiem nie była nowa linia, więc podnosi z powrotem semafor producenta. Producent opuszcza swój semafor, znów wypełnia pamięć dzieloną kolejnym znakiem i podnosi semafor pierwszego konsumenta. Pamięć wygląda tak:



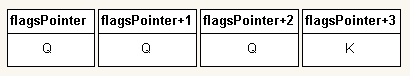
Załóżmy, że w tym momencie operator wysyła sygnał zatrzymania (SIGILL) do procesu P1. Każdy proces ma zaimplementowaną obsługę trzech sygnałów (SIGQUIT, SIGILL, SIGTRAP). Proces P1 uruchamia odpowiednią funkcję, która wysyła ten sam sygnał do procesu P2 i procesu P3. W efekcie wszystkie procesy zostają "zamrożone". Flagi stanu procesów zostają zmienione i pamięć dzielona stanu procesów wygląda teraz tak:



Flagi ustawione są na 'S' - stan stopu. W tym momencie procesy nie pracują, nie przyjmują żadnych danych, ale mogą obsłużyć sygnał wznowienia lub zakończenia działania. Sygnał stopu jest w tym momencie ignorowany. Chcemy aby program pracował dalej, więc wysyłamy sygnał wznowienia (SIGTRAP). Flagi stanu procesów zostają ponownie zmienione (znów pierwsze trzy ustawiane są na 'W' - stan pracy). Proces P2 dokańcza swoje operacje i podnosi semafor producenta.

Producent opuszcza swój semafor, do pamięci dzielonej zapisuje znak '\n' i uruchamia proces P2. Proces P2, inkrementuje zmienną przechowującą ilość znaków. Stwierdza, że odebrany znak jest nową linią, więc wysyła wynik do łącza nienazwanego i podnosi semafor konsumenta drugiego (dla procesu P3). Proces P3 opuszcza swój semafor, odbiera wynik, wypisuje na STDOUT i podnosi semafor producenta (dla procesu P1).

Proces P1 opuszcza swój semafor i oczekuje na wpisanie kolejnego ciągu znaków. Załóżmy, że wpisaliśmy znów jakiś ciąg znaków. Podczas obliczania ilości znaków wpisanego ciągu, wysyłamy sygnał zakończenia (SIGQUIT). Flagi stanów ponownie zostają zmienione:



Każdy proces uruchomi swoją funkcję kończącą. Procesy od razu zakończą się dzięki instrukcji "exit(0)". Na STDOUT nie zobaczymy wyniku, tylko napis "END OF ALL PROCESSES", który informuje nas, że wszystkie procesy zostały poprawnie zakończone a proces PM pomyślnie zwolnił wszystkie utworzone zasoby. Równie dobrze, najpierw moglibyśmy zatrzymać program a następnie wysłać sygnał zakończenia. Uzyskalibyśmy ten sam rezultat.

Program został wyposażony w menu, które ignoruje wszystkie inne wprowadzone opcje niż '1', '2' lub '3'. Sygnały można powielać, i wysyłać do różnych procesów w różnych kombinacjach. Po wysłaniu sygnału SIGQUIT nie można wysłać już innego sygnału - program musi zakończyć swoje działanie (zresztą nie zdążylibyśmy tego zrobić). Operator może zażądać pracy ze swoim plikiem lub z plikiem urandom. Jeżeli operator wpisze złą ścieżkę do pliku program będzie prosił o jej ponowne wprowadzenie.

Pamięć dzielona flag stanu procesów zapobiega np. wysłaniu 2 razy tego samego sygnału do danego procesu, ponieważ flaga mówi, jaki sygnał otrzymał dany proces (dodatkowo dla pewności sygnały są na czas swojej obsługi blokowane).

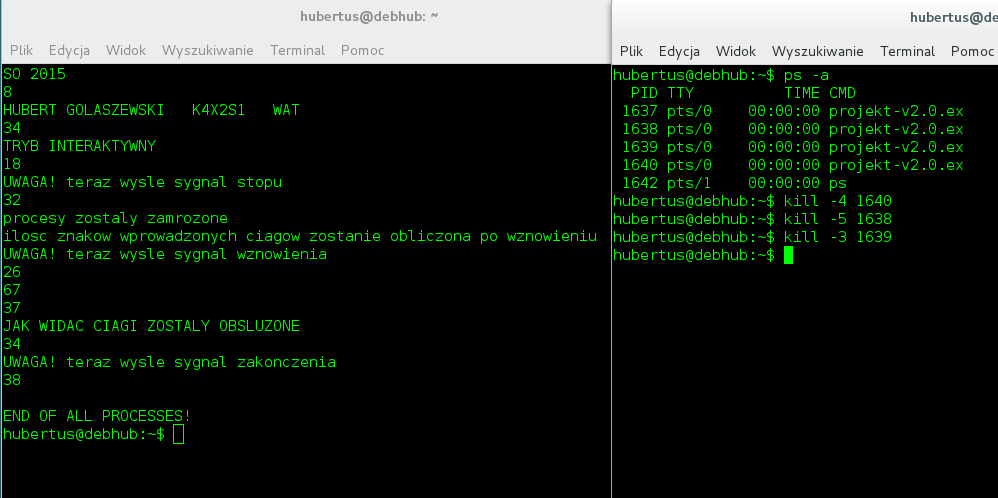
W programie zaimplementowana została obsługa błędów, więc można łatwo stwierdzić np. czy prawidłowo jest odłączana pamięć dzielona, semafory itp.

Flaga trybu właściwie pełni rolę kosmetyczną. Służy ona prawidłowemu wypisywaniu wyników na STDOUT (przy pracy z plikami w linii wypisywanych jest 10 jednostek, a w trybie interaktywnym 1 jednostka).

Semafor końca używany jest tylko przy kończeniu działania programu, kiedy zostaje zamykany proces P3 - podnosi on semafor końca (dla procesu PM) co pozwala na prawidłowe zwolnienie utworzonych zasobów.

**4. PREZENTACJA WYNIKÓW**

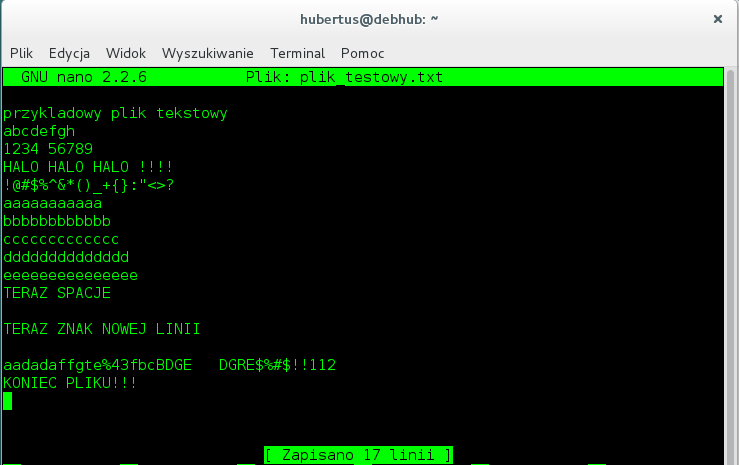
**TRYB INTERAKTYWNY**

****

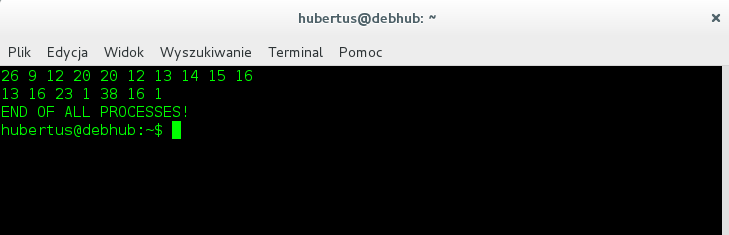
Na początku wpisałem kilka ciągów aby pokazać, że ilość znaków jest liczona poprawnie. Następnie do procesu P3 wysłałem sygnał zatrzymania. Można zauważyć, że po wpisaniu ciągów, wynik nie był wyświetlany. Po wysłaniu sygnału wznowienia do procesu P1 ciągi zostały obsłużone i program wypisał trzy wyniki pod rząd. Następnie wysłałem sygnał zakończenia do procesu P2 i program natychmiast zakończył działanie.

**TRYB PLIKU WŁASNEGO**

*ZAWARTOŚĆ TESTOWANEGO PLIKU:*

****

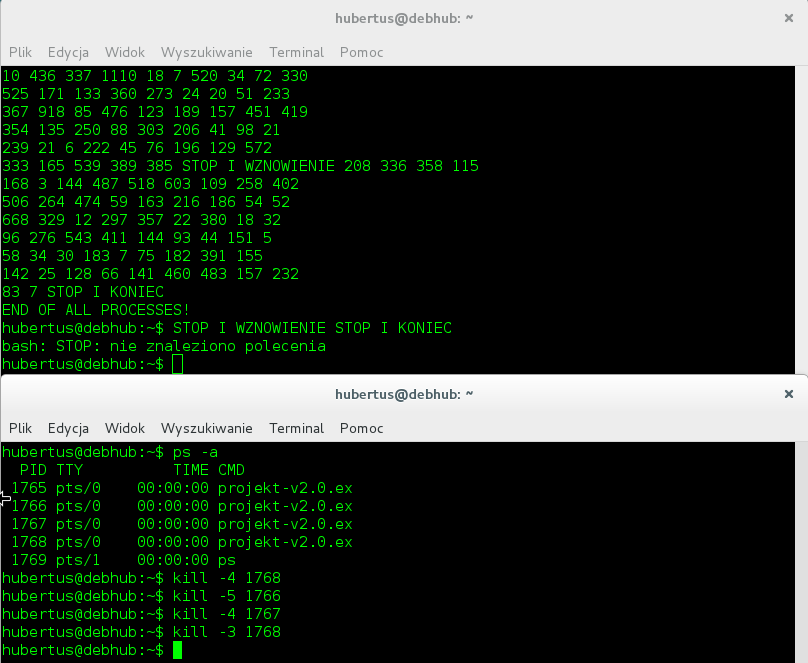
*WYNIK:*



Tym przykładem chciałem tylko pokazać, że tryb ten działa poprawnie. Wysyłanie sygnałów podczas obsługi tak małego pliku jest niewykonalne (program liczy za szybko).

W pliku zapisałem 17 linii i otrzymałem 17 wyników. W tym trybie mamy dwie możliwości zakończenia programu. Pierwszą opcją jest skończenie się pliku, a drugą możliwą do zrealizowania przy większym pliku jest wysłanie sygnału zakończenia.

**TRYB PLIKU URANDOM**

****

Ten przykład bardzo dobrze pokazuje jak działają sygnały. Najpierw wysłałem sygnał stopu do procesu P3. Procesy zostały zamrożone i mogłem napisać "STOP I WZNOWIENIE". Potem wysłałem sygnał wznowienia do procesu P1. Następnie wysłałem sygnał stopu do procesu P2. Napisałem "STOP I KONIEC" oraz wysłałem sygnał zakończenia do procesu P3. Program wypisał informację o zakończeniu programu i skończył działanie.

**5. WNIOSKI**

Implementacja projektu nie należała do prostych zadań, z uwagi na wiele szczegółów, które potrafiły zaważyć na poprawności działania całego programu. W moim projekcie wykorzystałem dwa obszary pamięci współdzielonej. Pierwszy przechowywał flagi stanu procesów a drugi dane przekazywane między procesem P1 a procesem P2. Do synchronizacji wymiany danych między procesami użyłem trzech semaforów nazwanych POSIX. Czwarty służy wyłącznie do prawidłowego zakończenia działania programu (proces macierzysty czeka nie wykonując pustych instrukcji). Komunikacja między procesem P2 a procesem P3 odbywała się za pomocą łącza nienazwanego (pipe).

Zdecydowałem się na przekazywanie ciągu z P1 do P2 po jednym znaku, gdyż takie rozwiązanie gwarantuje, że nie wystąpi przepełnienie bufora.

Program wyposażony jest w obsługę błędów (tworzenie semaforów, pamięci dzielonej, zwalnianie zasobów itp.).

Sygnał zakończenia to SIGQUIT, sygnał stopu SIGILL, a sygnał wznowienia SIGTRAP. Sygnały mogą być wysyłane do różnych procesów w różnych kombinacjach - to nie spowoduje błędu programu. Program został napisany tak, żeby po wysłaniu sygnału SIGQUIT nie przyjmował już żadnych sygnałów i od razu zakończył pracę. Programu nie da się również "zepsuć" podając złą opcję trybu czy nieprawidłową ścieżkę do pliku. Zastrzegam jednak prawo do kończenia programu sygnałem SIGKILL, ponieważ program nie zwalnia wtedy zasobów poprawnie i poprawność jego działania przy kolejnym uruchomieniu jest bynajmniej nieokreślona.

Możliwa jest praca w trybie interaktywnym, pracy z plikiem własnym lub plikiem urandom. Program w testach poradził sobie z plikiem tekstowym o rozmiarze 200 Mb, posiadającym w jednej linii nawet 140 tysięcy znaków. Z plikiem urandom pracował ponad 15 min, po czym sam zakończyłem jego działanie.

Prawidłowe zakończenie programu następuje tylko poprzez wysłanie sygnału zakończenia. Proces macierzysty zajmuje się zwalnianiem zasobów.

Program na pewno dałoby się napisać w o wiele krótszej wersji. Jednak ten sposób rozwiązania jest prosty do zrozumienia i mocno czytelny. Starałem się używać najprostszych funkcji i algorytmów.

Stwierdzam, że program w pełni realizuje założenia projektu i działa niezawodnie (nie da się doprowadzić do błędu programu operatorem z uprawnieniami określonymi w założeniach projektu). Program może być świetnym przykładem wymiany danych między procesami, pracy współbieżnej procesów, ich synchronizacji oraz obsługi sygnałów.