INSTYTUT TELEINFORMATYKI I AUTOMATYKI

Wydział Cybernetyki WAT

Przedmiot: SYSTEMY OPERACYJNE

SPRAWOZDANIE Z PROJEKTU

Temat: Zadanie projektowe – 11

Wykonał:

Karol Baranowski
K6X2S1

Data wykonania projektu:

22.01.2018 r.

Prowadzący:

mgr inż. Krystian Wojdygowski

1. Treść zadania projektowego

Zadanie projektowe - 11

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

- *Proces 1*: czyta wprowadzane dane (z pliku/klawiatury) i przekazuje je w niezmienionej formie do *procesu 2* poprzez mechanizm komunikacyjny **K1**.
- *Proces 2*: pobiera dane przesłane przez *proces 1*. Zlicza liczbę przesłanych znaków i przekazuje ją do *procesu 3* poprzez mechanizm komunikacyjny **K2**.
- *Proces 3*: pobiera dane wyprodukowane przez *proces 2* i wypisuje je na standardowym strumieniu diagnostycznym.

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 ÷ S3 należy wybrać samodzielnie spośród dostępnych w systemie.

W przygotowanym rozwiązaniu należy dodatkowo wykorzystać mechanizm synchronizacji (jeśli mechanizmy komunikacji tego wymagają).

Parametr	K1	K2
Mechanizm	pipe	pamięć współdzielona

2. Opis rozwiązania

Program działa zgodnie z poleceniem. Po jego włączeniu w procesie 1 użytkownik wybiera czy chce czytać dane z pliku czy wprowadzić je z klawiatury. Równolegle na drugim terminalu warto aby otworzył program do obsługi sygnałów. Użytkownik może wysłać sygnał do dowolnego z trzech procesów konsumenckich, a działania sygnałów są następujące: kończenie działania, wstrzymanie działania oraz wznowienie działania.

Wybór źródła w procesie 1 realizuje funkcja *ChooseInput()* zwracająca wskaźnik na plik. Następnie w nieskończonej pętli (nieskończone pętle występują we wszystkich procesach konsumenckich, jednak nie jest to nic szkodliwego, ponieważ poprzez mechanizmy synchronizacji oraz obsługę sygnałów zawsze można bezpiecznie wyjść z każdej bez błędów w programie) przepisywany jest podany tekst do odpowiedniego bufora (*data*) i wysyłany mechanizmem komunikacyjnym *pipe* do procesu 2, w którym następuje odbiór danych oraz zliczenie liczby ich znaków. Przy użyciu pipe nie było konieczności użycia mechanizmów synchronizacji procesów. W tym momencie do pamięci współdzielonej umieszczana jest liczba znaków, aby mógł ją odebrać proces 3, jednak w tym wypadku użycie semafor jest już konieczne, ponieważ proces 3 musi poczekać na odbiór do czasu, aż proces 2 wyśle dane do obszaru pamięci współdzielonej. W tym celu w obu tych procesach użyto funkcji *SemLock()* i *SemUnlock()* na odpowiednich id semafor. Proces 3 po odbiorze liczby znaków wyświetla ją na ekranie.

Po otrzymaniu sygnału kończenia działania program powróci do procesu macierzystego aby zamknąć wszystkie procesy, które zostały stworzone, łącza komunikacji: pipe i pamięć współdzieloną oraz semafory w celu zwolnienia zajętych zasobów.

3. Kod źródłowy programu

Kod modułu głównego:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <err.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#define rozmiar 500
#define PDES_READ 0
#define PDES_WRITE 1
#define STOP SIGINT
#define START SIGCONT
#define EXIT SIGTERM
bool Stop = false;
int Pids[3];
int ProcessID;
//SEMAFORY
union semun {
  int val;
  struct semid_ds * buf;
  unsigned short int * array;
  struct seminfo * __buf;
};
int SemLock(int semid) {
  struct sembuf opr;
  opr.sem_num = 0;
  opr.sem_op = -1;
  opr.sem_flg = 0;
  if (semop(semid, & opr, 1) == -1) {
    perror("Blad przy blokowaniu semafora. \n");
  } else {
    return 1;
  }
}
int SemUnlock(int semid) {
  struct sembuf opr;
  opr.sem_num = 0;
  opr.sem_op = 1;
  opr.sem_flg = 0;
  if (semop(semid, & opr, 1) == -1) {
    perror("Blad przy odblokowaniu semafora. \n");
  } else {
    return 1;
  }
}
int CreateSemaphore(char letter) {
    key_t key;
    int semid;
```

```
    union semun ctl;
    if ((key = ftok(".", letter)) == -1)
      perror("Blad przy tworzeniu klucza. \n");
    if ((semid = semget(key, 1, IPC_CREAT | 0600)) == -1)
      perror("Blad przy tworzeniu semafora. \n");
    ctl.val = 1;
    if (semctl(semid, 0, SETVAL, ctl) == -1)
      perror("Nie mozna ustawic semfora. \n");
    return semid;
  }
  //PAMIEC WSPOLDZIELONA
int CreateSharedMemory(size_t size) {
    key_t key; //unikalny
    int shmid; //int na id do shared memory
    if ((key = ftok(".", 'Q')) == -1)
      perror("Blad utworzenia klucza. \n");
    if ((shmid = shmget(key, size, IPC_CREAT | 0666)) < 0)
      perror("Blad tworzenia segmentu. \n");
    return shmid; //id segmentu pamieci
  }
  //ODCZYT
FILE * ChooseInput() {
    int wybor;
    printf("Witaj w procesie 1. \n");
    printf("Proces pobiera dane z tekstu wpisanego badz pliku.Wybierz: \n");
    printf("1 - aby pobrac z tekstu wpisanego \n");
    printf("2 - aby pobrac z pliku \n");
    wybor = getchar() - '0';
    if (wybor == 1) {
      return stdin;
    }
    FILE * input;
    char name[150];
    printf("Nazwa pliku: ");
    fscanf(stdin, "%s", name);
    input = fopen(name, "r");
    if (input == NULL)
      perror("Nie ma pliku. \n");
    return input;
  }
  //SYGNALY
void SetHandler(sighandler_t handler) {
  struct sigaction newAction;
  newAction.sa_handler = handler;
  sigaction(STOP, & newAction, NULL);
  sigaction(START, & newAction, NULL);
  sigaction(EXIT, & newAction, NULL);
}
void Handler(int sig) {
  if (ProcessID == 1) {
    if (sig == STOP)
      Stop = true;
    else if (sig == START)
      Stop = false;
```

```
    else if (sig == EXIT)
      exit(0);
  } else {
    kill(Pids[0], sig);
    if (sig == EXIT) {
      exit(0);
    }
  }
}
void p1(int pipes[]) {
  sleep(1);
  ProcessID = 1;
  close(pipes[PDES_READ]);
  char data[rozmiar] = {};
  FILE * input = ChooseInput();
  while (1) {
    if (Stop) {
      sleep(1);
      continue;
    }
    if (fgets(data, rozmiar, input) != NULL) {
      write(pipes[PDES_WRITE], data, rozmiar);
    }
    //   else
    //       {
    //           fclose(input);
    //       }
  }
}
void p2(int pipes[], int semid1, int semid2, int shmid) {
  ProcessID = 2;
  close(pipes[PDES_WRITE]);
  char data[rozmiar];
  //char liczba[rozmiar];
  char * shm = shmat(shmid, NULL, 0); //podlaczenie do shm
  while (1) {
    SemLock(semid1);
    memset(data, 0, rozmiar);
    read(pipes[PDES_READ], data, rozmiar);
    data[0] = (char)(strlen(data) - 1);
    memcpy(shm, data, rozmiar);
    SemUnlock(semid2);
  }
}
void p3(int shmid, int semid1, int semid2) {
  ProcessID = 3;
  char * shm;
  char data[rozmiar];
  shm = shmat(shmid, NULL, 0);
  while (1) {
    SemLock(semid2);
    memcpy(data, shm, rozmiar);
    fprintf(stderr, "%d", data[0]);
    putchar('\n');
```

```
    SemUnlock(semid1);
  }
}
void SavePids() {
    FILE * file = fopen("data", "w");
    if (!file) {
      perror("data");
      exit(1);
    }
    fprintf(file, "PPID:%d PID1: %d PID2: %d PID3: %d \n", getpid(), Pids[0], Pids[1], Pids[2]);
    fclose(file);
  }
  //****************************
  //****************************
int main() {
  int shmid = CreateSharedMemory(rozmiar);
  int semid1 = CreateSemaphore('b');
  int semid2 = CreateSemaphore('c');
  SemLock(semid2);
  //PIPES
  int pipeDes[2];
  pipe(pipeDes);
  if (pipe(pipeDes) == -1) {
    perror("Blad przy tworzeniu pipe. \n");
  }
  SetHandler(Handler);
  if ((Pids[0] = fork()) == 0) {
    printf("Proces konsumenta 1 PID:%d PPID: %d \n", getpid(), getppid());
    p1(pipeDes);
  }
  if ((Pids[1] = fork()) == 0) {
    printf("Proces konsumenta 2 PID:%d PPID: %d \n", getpid(), getppid());
    p2(pipeDes, semid1, semid2, shmid);
  }
  if ((Pids[2] = fork()) == 0) {
    printf("Proces konsumenta 3 PID:%d PPID: %d \n", getpid(), getppid());
    p3(shmid, semid1, semid2);
  }
  SavePids();
  wait(NULL);
  for (int i = 0; i < 3; i++) {
    kill(Pids[i], SIGKILL);
  }
  shmctl(shmid, IPC_RMID, NULL);
  semctl(semid1, 0, IPC_RMID);
  semctl(semid2, 0, IPC_RMID);
  return 0;
}
```

Kod panelu do wysyłania sygnałów:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#define STOP SIGINT
#define START SIGCONT
#define EXIT SIGTERM
int Pids[3];
int ParentPid;
void LoadPids() {
  FILE * file = fopen("data",  "r");
  if (!file) {
    perror("data");
    exit(1);
  }
  fscanf(file,  "PPID:%d PID1: %d PID2: %d PID3: %d \n", & ParentPid, & Pids[0], & Pids[1], & Pids[2]);
  fclose(file);
}
int main() {
  LoadPids();
  int nr = 1;
  int syg;
  while (1) {
    printf("Proces macierzysty: %d . Wybierz proces do wyslania sygnalu:  \n", ParentPid);
    printf("1.Proces 1 %d \n", Pids[0]);
    printf("2.Proces 2 %d \n", Pids[1]);
    printf("3.Proces 3 %d \n", Pids[2]);
    printf("4.Wyjdz z programu \n \n");
    scanf("%d", & nr);
    if (nr == 4)
      exit(0);
    else if (nr != 1 && nr != 2 && nr != 3) {
      printf("Niepoprawny wybor numeru procesu. Sprobuj ponownie");
      continue;
    }
    nr--;
    printf("Lista sygnalow:. \n1. Wstrzymaj dzialanie; \n2. Wznow dzialanie; \n3. Zakoncz dzialanie. \n \n");
    printf("Wybierz sygnal: 1 - 3  \n");
    scanf("%d", & syg);
    switch (syg) {
    case 1:
      kill(Pids[nr], STOP);
      break;
    case 2:
      kill(Pids[nr], START);
      break;
    case 3:
      kill(Pids[nr], EXIT);
      break;
    default:
      printf("Niepoprawny wybor sygnalu.Sprobuj ponownie \n");
      break;
    }
```

```
    }
    return 0;
}
```

4. Wyniki uruchomienia programu

Przykładowy zrzut ekranu z paroma funkcjonalnościami programu: