Билет 4/5 (Пример из ЛР Производство-потребление)

```
#define P -1
#define V +1

#define SB 0
#define SE 1
#define SF 2

typedef struct sembuf sembuf;

sembuf take_semafor_producer[2] =
{
    { SE, P, SEM_UNDO },
    { SB, P, SEM_UNDO }
};

sembuf free_semafor_producer[2] =
{
    { SB, V, SEM_UNDO },
    { SF, V, SEM_UNDO },
};

sembuf take_semafor_consumer[2] =
{
    { SF, P, SEM_UNDO },
    { SB, P, SEM_UNDO }
};

sembuf free_semafor_consumer[2] =
{
    { SB, V, SEM_UNDO },
    { SE, V, SEM_UNDO }
};

void producer(int* shared_bufer, const int len, const int semid)
{
    while(1)
    {
        sleep(1);

        if (semop(semid, take_semafor_producer, 2) == -1)
            semop_error();

       *(shared_bufer + *shared_bufer) = *shared_bufer - 1;

       printf("Produser [\%d] --> \%d\n", getpid(), *(shared_bufer + *
          shared_bufer));

       (*shared_bufer)++;

       if (semop(semid, free_semafor_producer, 2) == -1)
           semop_error();

       if (*shared_bufer >= len - 1)
       {
           *shared_bufer = 2;
       }
    }

    exit(0);
}

void consumer(int* shared_bufer, const int len, const int semid)
{
    while(1)
    {
        sleep(1);

        if (semop(semid, take_semafor_consumer, 2) == -1)
            semop_error();

        printf("Consumer [\%d] <-- \%d\n", getpid(), *(shared_bufer + *(
           shared_bufer + 1)));

        (*(shared_bufer + 1))++;

        if (semop(semid, free_semafor_consumer, 2) == -1)
            semop_error();

        if (*(shared_bufer + 1) >= len - 1)
        {
            *(shared_bufer + 1) = 2;
        }
    }

    exit(0);
}

#define COUNT 3
#define N 8
#define PERM  S_IRWXU | S_IRWXG | S_IRWXO

int main(void)
{
    pid_t producers[COUNT];
    pid_t consumers[COUNT];
    int status, shmid, semid, ctrl_sb, ctrl_se, ctrl_sf, i;
    int* shared_bufer;

    shmid = shmget(IPC_PRIVATE, (N + 2) * sizeof(int), IPC_CREAT | PERM);

    if (shmid == -1)
        shget_error();

    shared_bufer = shmat(shmid, 0, 0);

    if (*((int*)shared_bufer) == -1)
        shmat_error();

    semid = semget(IPC_PRIVATE, 3, IPC_CREAT | PERM);

    if (semid == -1)
        semget_error();

    ctrl_sb = semctl(semid, SB, SETVAL, 1);
    ctrl_se = semctl(semid, SE, SETVAL, N);
    ctrl_sf = semctl(semid, SF, SETVAL, 0);

    if (ctrl_sb == -1 || ctrl_se == -1 || ctrl_sf == -1)
        semctl_error();

    *shared_bufer = 2;
    *(shared_bufer + 1) = 2;

    for (i = 0; i < COUNT; ++i)
    {
        producers[i] = fork();

        if (producers[i] == -1)
            fork_error();
        else if (producers[i] == 0)
            producer(shared_bufer, N, semid);

        consumers[i] = fork();

        if (consumers[i] == -1)
            fork_error();
        else if (consumers[i] == 0)
            consumer(shared_bufer, N, semid);

        sleep(2);
    }

    for (i = 0; i < COUNT; ++i)
    {
        wait(&status);
        wait(&status);
    }

    if (shmctl(shmid, IPC_RMID, NULL) == -1)
        clean_error();

    if (semctl(semid, 0, IPC_RMID, 0) == -1)
        clean_error();

    return 0;
}
```

Билет 6/19 (Пример из ЛР Читатели-пистаели Win32 API)

```
#include <stdio.h>
#include <windows.h>
#include <iostream>
using namespace std;

#define OK 0
#define ERROR 1

#define WRITERS 3
#define READERS 5
#define ITERS 5

int val = 0;
bool activewriter = false;
int readercount = 0;

int waiting_writers = 0;
int waiting_readers = 0;

HANDLE writers[WRITERS];
HANDLE readers[READERS];
HANDLE can_write;
HANDLE can_read;

void start_read() {
    InterlockedIncrement(&waiting_readers);
    if (true == activewriter || waiting_writers > 0) {
        WaitForSingleObject(can_read, INFINITE);
    }
    InterlockedDecrement(&waiting_readers);
    InterlockedIncrement(&readercount);
    SetEvent(can_read);
}

void stop_read() {
    InterLockedDecrement(readercount);
    if (0 == readercount) {
        SetEvent(can_write);
    }
}

void start_write() {
    InterlockedIncrement(&waiting_writers);
    if (readercount > 0 || true == activewriter) {
        WaitForSingleObject(can_write, INFINITE);
    }
    InterlockedDecrement(&waiting_writers);
    activewriter = true;
}

void stop_write() {
    activewriter = false;
    ResetEvent(can_write);
    if (waiting_readers > 0) {
        SetEvent(can_read);
    } else {
        SetEvent(can_write);
    }
}

DWORD WINAPI writer(LPVOID) {
    for (int i = 0; i < ITERS; i++) {
        start_write();
        val++;
        cout << "Writer" << GetCurrentThreadId() << " write " << val << endl;
        stop_write();
        Sleep(100);
    }
    return OK;
}

DWORD WINAPI reader(LPVOID mutex) {
    for (int i = 0; i < ITERS + 7; i++) {
        start_read();
        WaitForSingleObject(mutex, INFINITE);
        cout << "Reader" << GetCurrentThreadId() << " read " << val << endl;
        ReleaseMutex(mutex);
        stop_read();
        Sleep(100);
    }
    return OK;
}

int create_mutex_threads() {
    HANDLE mutex = CreateMutex(NULL, FALSE, NULL);
    if (NULL == mutex) {
        cout << "Can't create mutex\n";
        return ERROR;
    }
    // создание писателей
    for (int i = 0; i < WRITERS; i++) {
        // данный аргумент определяет, может ли создаваемый поток быть унаследова
           н дочерним процессом
        // размер стека в байтах. Если передать 0, то будет использоваться значен
           ие по - умолчанию (1 мегабайт)
        // адрес функции, которая будет выполняться потоком
        // указатель на переменную, которая будет передана в поток
        // флаги создания
        // указатель на переменную, куда будет сохранён идентификатор потока

        writers[i] = CreateThread(NULL, 0, &writer, NULL, 0, NULL);
        if (NULL == writers[i]) {
            cout << "Can't create threads\n";
            return ERROR;
        }
    }
    // создание читателей
    for (int i = 0; i < READERS; i++) {
        readers[i] = CreateThread(NULL, 0, &reader, mutex, 0, NULL);
        if (NULL == readers[i]) {
            cout << "Can't create threads\n";
            return ERROR;
        }
    }

    return OK;
}

int create_events() {
    // атрибут защиты
    // тип сброса TRUE - ручной
    // начальное состояние TRUE - сигнальное
    // имя объекта

    // с автосбросом
    can_read = CreateEvent(NULL, FALSE, FALSE, TEXT("ReadEvent"));
    if (can_read == NULL) {
        cout << "Can't create event\n";
        return ERROR;
    }

    // с ручным сбросом
    can_write = CreateEvent(NULL, TRUE, FALSE, TEXT("WriteEvent"));
    if (can_write == NULL) {
        cout << "Can't create event\n";
        return ERROR;
    }
    return OK;
}

int main() {

    if (create_events() != OK) {
        return ERROR;
    }
    if (create_mutex_threads() != OK) {
        return ERROR;
    }

    WaitForMultipleObjects(WRITERS, writers, TRUE, INFINITE);
    WaitForMultipleObjects(READERS, readers, TRUE, INFINITE);

    return OK;
}
```

Билет 11 (очереди сообщений и программные каналы – сравнение, примеры (для программных каналов пример из лабораторной работы с сигналами).)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <signal.h>

#define ANSI_COLOR_GREEN "\x1b[32m"
#define ANSI_COLOR_RESET "\x1b[0m"

#define N 20

void call_signal_int()
{
    printf("Child exit\n");
    exit(0);
}

int main()
{
    int status;
    pid_t childpid1, childpid2;
    char buffer[N];
    char message1[N] = "message child1";
    char message2[N] = "message child2";
    int channel1[2], channel2[2];

    if (pipe(channel1) == -1)
    {
        perror("Cant't pipe.\n");
        return 1;
    }

    childpid1 = fork();

    if (childpid1 == -1)
    {
        perror("Cant' fork.\n");
        return 1;
    }
    else if (childpid1 == 0)
    {
        signal(SIGINT, call_signal_int);

        close(channel1[0]);
        write(channel1[1], message1, sizeof(message1));
        sleep(2);
        exit(0);
    }

    if (pipe(channel2) == -1)
    {
        perror("Cant't pipe.\n");
        return 1;
    }

    childpid2 = fork();

    if (childpid2 == -1)
    {
        perror("Can't fork.\n");
        return 1;
    }
    else if (childpid2 == 0)
    {
        signal(SIGINT, call_signal_int);

        close(channel2[0]);
        write(channel2[1], message2, sizeof(message2));
        sleep(2);
        exit(0);
    }
    else
    {
        signal(SIGINT, SIG_IGN);
        wait(&status);
        if (WIFEXITED(status))
        {
            printf(ANSI_COLOR_GREEN "child process exit success\n"
                ANSI_COLOR_RESET);

            close(channel1[1]);
            read(channel1[0], buffer, sizeof(buffer));
            printf("message1 = \%s\n", buffer);

            close(channel2[1]);
            read(channel2[0], buffer, sizeof(buffer));
            printf("message2 = \%s\n", buffer);
            return 0;
        }
    }

    return 0;
}
```

Билет 16 (Пример из ЛР fork, exec, wait, signal)

При завершении любого процесса система проверяет не осталось ли у него незавершенных потомков (по дескриптору процесса, где есть указатели на потомков). Если такие остались, то выполняется процесс усыновления, фактически изменяя указатели: процесс-потомок получает указатель на нового процесса-предка.

```
int main(void)
{
  pid_t childpid1;
  pid_t childpid2;
  if ((childpid1 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди
     тельском процессе */
  {
    perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли
       бо */
    exit(1);                                        /* таблица заполнена) */
  }
  if (childpid1 == 0)
  { /* здесь располагается дочерний код */
    printf("Child1 forked \n");
    printf("Child1 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
        (), getpgrp());
    getchar();
    printf("Child1 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
        (), getpgrp());
    return 0;
  }

  if ((childpid2 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди
     тельском процессе */
  {
    perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли
       бо */
    exit(1);                                        /* таблица заполнена) */
  }
  if (childpid2 == 0)
  { /* здесь располагается дочерний код */
    printf("Child2 forked \n");
    printf("Child2 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
        (), getpgrp());
    getchar();
    printf("Child2 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
        (), getpgrp());
  }

  if (childpid1 != 0 && childpid2 != 0)
  { /* здесь располагается родительский код */
    printf("Parent id = %d, child1 id = %d, child2 id = %d, group id = %d \n",
       getpid(), childpid1, childpid2, getpgrp());
    getchar();
    printf("Parent exited \n");
  }

  return 0;
}
```

Системный вызов wait(&status) вызывается в теле предка. Системный вызов wait() блокирует родительский процесс до момента завершения дочернего. При их завершении предок получает статус завершения потомка.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  pid_t childpid1;
  pid_t childpid2;
  if ((childpid1 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди

     тельском процессе */
  {
    perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли
       бо */
    exit(1);                                        /* таблица заполнена) */
  }
  if (childpid1 == 0)
  { /* здесь располагается дочерний код */
    printf("Child1 forked \n");
    printf("Child1 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
        (), getpgrp());
    getchar();
    printf("Child1 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
        (), getpgrp());
    return 0;
  }

  if ((childpid2 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди
     тельском процессе */
  {
    perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли
       бо */
    exit(1);                                        /* таблица заполнена) */
  }
  if (childpid2 == 0)
  { /* здесь располагается дочерний код */
    printf("Child2 forked \n");
    printf("Child2 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
        (), getpgrp());
    getchar();
    printf("Child2 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
        (), getpgrp());
  }

  if (childpid1 != 0 && childpid2 != 0)
  { /* здесь располагается родительский код */
    printf("Parent id = %d, child1 id = %d, child2 id = %d, group id = %d \n",
       getpid(), childpid1, childpid2, getpgrp());
    int status1;
    childpid1 = wait(&status1);
    printf("Child has finished: PID = %d\n", childpid1);

    int status2;
    childpid2 = wait(&status2);
    printf("Child has finished: PID = %d\n", childpid2);

    if (WIFEXITED(status1) && WIFEXITED(status2))
      printf("Children exited with code %d and %d\n", WEXITSTATUS(status1),
         WEXITSTATUS(status2));
    else
      printf("Child terminated abnormally\n");
  }
  return 0;
}
```

Чаще всего нет смысла в выполнении двух одинаковых процессов и потомок сразу выполняет системный вызов exec(), параметрами которого является имя исполняемого файла и, если нужно, параметры, которые будут переданы этой программе. Говорят, что системный вызов exec() создает низкоуровневый процесс: создаются таблицы страниц для адресного пространства программы, указанной в exec(), но программа на выполнение не запускается, так как это не полноценный процесс, имеющий идентификатор и дескриптор. Системный вызов exec() создает таблицу страниц для адресного пространства программы, переданной ему в качестве параметра, а затем заменяет старый адрес новой таблицы страниц.

Бывает шести видов: execlp, execvp, execl, execv, execle, execve.

В результате системного вызова exec() адресное пространство процесса будет заменено на адресное пространство новой программы, а сам процесс будет возвращен в режим задачи с установкой указателя команд на первую выполняемую инструкцию этой программы.

Возникает процесс-зомби – это процесс, у которого отобраны все ресурсы, кроме последнего – строки в таблице процессов. Если потомок завершился(аварийно завершился exec) до вызовы wait у предка, то для того чтобы предок не завис, возникает зомби. Зомби существует пока не будет выполнен wait у родителя.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  pid_t childpid1;
  pid_t childpid2;
  if ((childpid1 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди
     тельском процессе */
  {
    perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли
       бо */
    exit(1);                                        /* таблица заполнена) */
  }
  if (childpid1 == 0)
  { /* здесь располагается дочерний код */
    printf("Child1 forked \n");
    printf("Child1 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
        (), getpgrp());

    char msg[] = "exec() called from child1";

    /* обращаемся к программе echo, параметрами передаём её имя и текст для печат
       и */
    /* NULL - завершает список параметров */
        if (execlp("echo", "echo", msg, NULL) == -1) /* при успешном вызове exec
            () заменяет адресное пространство процесса */
        {
            perror("Can't exec"); /* - эти вызовы произойдут только в том случае,
                если exec() не выполнится*/
      exit(1);
        }
    return 0;
  }

  if ((childpid2 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди
     тельском процессе */
  {
    perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли
       бо */
    exit(1);                                        /* таблица заполнена) */
  }
  if (childpid2 == 0)
  { /* здесь располагается дочерний код */
    printf("Child2 forked \n");
    printf("Child2 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
        (), getpgrp());

    char msg[40] = "exec() called from child2";

    /* обращаемся к программе echo, параметрами передаём её имя и текст для печат
       и */
    /* NULL - завершает список параметров */
    if (execlp("echo", "echo", msg, NULL) == -1) /* при успешном вызове exec() за
       меняет адресное пространство процесса */
        {
            perror("Can't exec"); /* - эти вызовы произойдут только в том случае,
                если exec() не выполнится*/

            exit(1);
        }
  }

  if (childpid1 != 0 && childpid2 != 0)
  { /* здесь располагается родительский код */
    printf("Parent id = %d, child1 id = %d, child2 id = %d, group id = %d \n",
       getpid(), childpid1, childpid2, getpgrp());
    int status1;
    childpid1 = wait(&status1);
    printf("Child has finished: PID = %d\n", childpid1);

    int status2;
    childpid2 = wait(&status2);
    printf("Child has finished: PID = %d\n", childpid2);

    if (WIFEXITED(status1) && WIFEXITED(status2))
      printf("Children exited with code %d and %d\n", WEXITSTATUS(status1),
         WEXITSTATUS(status2));
    else
      printf("Child terminated abnormally\n");
  }
  return 0;
}
```

Сигнал - способ информировании процесса ядром о происшествии какого-то события. Если возникает несколько одинотипных событий, процессу будет подан только один сигнал. Сигнал означает, что произошло событие, но ядро не сообщает сколько таких событий произошло.

Средство передачи сигнала – kill(), приема сигнала – signal().

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

int signal_flag = 0;

/* собственный обработчик сигнала ctrl-c */
void catch_sigp(int sig_numb)
{
    signal(sig_numb, catch_sigp);
    signal_flag = 1;
    printf("\n Parent catch sig %d\n", sig_numb);
}

int main(void)
{
  pid_t childpid1;
  pid_t childpid2;
  /* дескрипторы программных каналов */
  int my_pipe1[2];
  int my_pipe2[2];
  /* [0] - выход для чтения, [1] - выход для записи */

    /* потомок унаследует открытый программный канал предка */
  if (pipe(my_pipe1) == -1)
  {
    perror("Can't pipe");
    exit(1);
  }
  if (pipe(my_pipe2) == -1)
  {
    perror("Can't pipe");
    exit(1);
  }

if ((childpid1 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди
   тельском процессе */
{
  perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли
     бо */
  exit(1);                                        /* таблица заполнена) */
}
if (childpid1 == 0)
{ /* здесь располагается дочерний код */
  printf("Child1 forked \n");
  printf("Child1 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
      (), getpgrp());

  char buf[40];
  close(my_pipe1[1]); /* потомок ничего не запишет в канал */
  read(my_pipe1[0], buf, sizeof(buf));
  printf("Child1 catch message = %s \n", buf);

  return 0;
}
```

```
  if ((childpid2 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди
     тельском процессе */
  {
    perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли
       бо */
    exit(1);                                    /* таблица заполнена) */
  }
  if (childpid2 == 0)
  { /* здесь располагается дочерний код */
    printf("Child2 forked \n");
    printf("Child2 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
       (), getpgrp());

    char buf[40];
        close(my_pipe2[1]); /* потомок ничего не запишет в канал */
    read(my_pipe2[0], buf, sizeof(buf));
    printf("Child2 catch message = %s \n", buf);
  }

  if (childpid1 != 0 && childpid2 != 0)
  { /* здесь располагается родительский код */
    printf("Parent id = %d, child1 id = %d, child2 id = %d, group id = %d \n",
       getpid(), childpid1, childpid2, getpgrp());

    /* ожидание сигнала */
    signal(SIGINT, catch_sigp);
    sleep(2);

    if (!signal_flag)
    {
      char msg1[] = "Hello, child1!";
      close(my_pipe1[0]); /* предок ничего не считает из канала */
      write(my_pipe1[1], msg1, sizeof(msg1));

      char msg2[] = "Hello, child2!";
      close(my_pipe2[0]); /* предок ничего не считает из канала */
      write(my_pipe2[1], msg2, sizeof(msg2));
    }

    int status1;
    childpid1 = wait(&status1);

     printf("Child has finished: PID = %d\n", childpid1);

     int status2;
     childpid2 = wait(&status2);
     printf("Child has finished: PID = %d\n", childpid2);

     if (WIFEXITED(status1) && WIFEXITED(status2))
       printf("Children exited with code %d and %d\n", WEXITSTATUS(status1),
          WEXITSTATUS(status2));
     else
       printf("Child terminated abnormally\n");
  }
  return 0;
}
```

Билет 15 (Пример из ЛР Сигналы, Программные каналы)

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

int signal_flag = 0;

/* собственный обработчик сигнала ctrl-c */
void catch_sigp(int sig_numb)
{
    signal(sig_numb, catch_sigp);
    signal_flag = 1;
    printf("\n Parent catch sig %d\n", sig_numb);
}

int main(void)
{
  pid_t childpid1;
  pid_t childpid2;
  /* дескрипторы программных каналов */
  int my_pipe1[2];
  int my_pipe2[2];
  /* [0] - выход для чтения, [1] - выход для записи */

    /* потомок унаследует открытый программный канал предка */
  if (pipe(my_pipe1) == -1)
  {
    perror("Can't pipe");
    exit(1);
  }
  if (pipe(my_pipe2) == -1)
{
  perror("Can't pipe");
  exit(1);
}

if ((childpid1 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди
   тельском процессе */
{
  perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли
     бо */
  exit(1);                                          /* таблица заполнена) */
}
if (childpid1 == 0)
{ /* здесь располагается дочерний код */
  printf("Child1 forked \n");
  printf("Child1 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
     (), getpgrp());

  char buf[40];
  close(my_pipe1[1]); /* потомок ничего не запишет в канал */
  read(my_pipe1[0], buf, sizeof(buf));
  printf("Child1 catch message = %s \n", buf);

  return 0;
}

if ((childpid2 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди
   тельском процессе */
{
  perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли
     бо */
  exit(1);                                          /* таблица заполнена) */
}
if (childpid2 == 0)
{ /* здесь располагается дочерний код */
  printf("Child2 forked \n");
  printf("Child2 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
     (), getpgrp());

  char buf[40];
      close(my_pipe2[1]); /* потомок ничего не запишет в канал */
  read(my_pipe2[0], buf, sizeof(buf));
  printf("Child2 catch message = %s \n", buf);
}

if (childpid1 != 0 && childpid2 != 0)
{ /* здесь располагается родительский код */
  printf("Parent id = %d, child1 id = %d, child2 id = %d, group id = %d \n",
     getpid(), childpid1, childpid2, getpgrp());

  /* ожидание сигнала */
  signal(SIGINT, catch_sigp);
  sleep(2);

  if (!signal_flag)
  {
    char msg1[] = "Hello, child1!";
    close(my_pipe1[0]); /* предок ничего не считает из канала */
    write(my_pipe1[1], msg1, sizeof(msg1));

    char msg2[] = "Hello, child2!";
    close(my_pipe2[0]); /* предок ничего не считает из канала */
    write(my_pipe2[1], msg2, sizeof(msg2));


    }

    int status1;
    childpid1 = wait(&status1);
    printf("Child has finished: PID = %d\n", childpid1);

    int status2;
    childpid2 = wait(&status2);
    printf("Child has finished: PID = %d\n", childpid2);

    if (WIFEXITED(status1) && WIFEXITED(status2))
      printf("Children exited with code %d and %d\n", WEXITSTATUS(status1),
         WEXITSTATUS(status2));
    else
      printf("Child terminated abnormally\n");
  }
  return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

#define ANSI_COLOR_GREEN "\x1b[32m"
#define ANSI_COLOR_RESET "\x1b[0m"

#define N 20

int main()
{


    int status;
    pid_t childpid1, childpid2;
    char buffer[N];
    char message1[N] = "hello";
    char message2[N] = "goodbye";
    int channel1[2], channel2[2];

    if (pipe(channel1) == -1)
    {
        perror("Can not pipe.\n");
        return 1;
    }

    childpid1 = fork();

    if (childpid1 == -1)
    {
        perror("Can not fork.\n");
        return 1;
    }
    else if (childpid1 == 0)
    {
        close(channel1[0]);
        write(channel1[1], message1, sizeof(message1));
        exit(0);
    }

    if (pipe(channel2) == -1)
    {
        perror("Can not pipe.\n");
        return 1;
    }

    childpid2 = fork();

    if (childpid2 == -1)
    {
        perror("Can not fork.\n");
        return 1;
    }
    else if (childpid2 == 0)
    {
        close(channel2[0]);
        write(channel2[1], message2, sizeof(message2));
        exit(0);
    }
    else
    {
        wait(&status);
        if (WIFEXITED(status))
        {
            printf(ANSI_COLOR_GREEN "child process exit success\n"
               ANSI_COLOR_RESET);

            close(channel1[1]);
            read(channel1[0], buffer, sizeof(buffer));
            printf("message = %s\n", buffer);

            close(channel2[1]);
            read(channel2[0], buffer, sizeof(buffer));
            printf("message = %s\n", buffer);
        }
    }


        return 0;
    }
```

Билет 9/19 (Пример из ЛР Процессе демоны и сирота Unix)

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

#define ANSI_COLOR_GREEN "\x1b[32m"
#define ANSI_COLOR_RESET "\x1b[0m"

int main()
{
    int status;
    pid_t childpid1, childpid2;

    childpid1 = fork();

    if (childpid1 == -1)
    {
        perror("Can't fork.\n");

      return 1;
   }
   else if (childpid1 == 0)
   {
       if (execl("/bin/ls", "ls", "-lah", 0) == -1)
       {
           perror("Can't exec.\n");
           exit(1);
       }
   }

   childpid2 = fork();

   if (childpid2 == -1)
   {
       perror("Can't fork.\n");
       return 1;
   }
   else if (childpid2 == 0)
   {
       if (execl("/bin/cat", "cat", "makefile", 0) == -1)
       {
           perror("Can't exec.\n");
           exit(1);
       }
   }
   else
   {
       wait(&status);

       if (WIFEXITED(status)) printf(ANSI_COLOR_GREEN "child process exit
          success\n" ANSI_COLOR_RESET);
   }

   return 0;
}
```

```c
#include <stdio.h>
#include <sys/types.h>

int main()
{
    pid_t childpid1, childpid2;

    childpid1 = fork();

    if (childpid1 == -1)
    {
        perror("Can't fork.\n");
        return 1;
    }
    else if (childpid1 == 0)
    {
        printf("CHILD1\npid: \%d; ppid: \%d; gid: \%d\n\n", getpid(), getppid(),
           getgid());
        getchar();
        printf("CHILD1\npid: \%d; ppid: \%d; gid: \%d\n\n", getpid(), getppid(),


           getgid());
        return 0;
    }

    childpid2 = fork();

    if (childpid2 == -1)
    {
        perror("Can't fork.\n");
        return 1;
    }
    else if (childpid2 == 0)
    {
        printf("CHILD2\npid: \%d; ppid: \%d; gid: \%d\n\n", getpid(), getppid(),
           getgid());
        getchar();
        printf("CHILD2\npid: \%d; ppid: \%d; gid: \%d\n\n", getpid(), getppid(),
           getgid());
        return 0;
    }
    else
    {
        printf("PARENT\npid: \%d; ppid: \%d; gid: \%d\n\n", getpid(), getppid(),
           getgid());
        getchar();
    }

    return 0;
}
```

Билет 8/12/20/21/24 (Пример из ЛР Производство-потребление и читатели-писатели Unix)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <stdio.h>
#include <sys/shm.h>
#include <signal.h>

#define COUNT 3
#define PRODUCER 0
#define CONSUMER 1

#define PERMS S_IRWXU | S_IRWXG | S_IRWXO

#define EMPTYCOUNT 0
#define FULLCOUNT 1
#define BIN 2

int semaphore;
int shared_memory;
char **addr_shared_memory;

// Массив структур
struct sembuf producer_grab[2] = { {EMPTYCOUNT, -1, SEM_UNDO}, {BIN, -1, SEM_UNDO
    } };
struct sembuf producer_free[2] = { {BIN, 1, SEM_UNDO}, {FULLCOUNT, 1, SEM_UNDO}
    };
struct sembuf consumer_grab[2] = { {FULLCOUNT, -1, SEM_UNDO}, {BIN, -1, SEM_UNDO}
    };
struct sembuf consumer_free[2] = { {BIN, 1, SEM_UNDO}, {EMPTYCOUNT, 1, SEM_UNDO}
    };

// Потребитель
void consumer(int semaphore, int value)
{
  while(1)
  {
    sleep(1);
    int sem_op_p = semop(semaphore, consumer_grab, 2);
    if (sem_op_p == -1)
    {
        perror("Can't semop \\n");
        exit(1);
    }

    if ((char*)(*(addr_shared_memory + sizeof(int *))) == ((char*)(
        addr_shared_memory) + 2 * sizeof(int *) + 5 * sizeof(int)))
      *(addr_shared_memory + sizeof(int *)) = (char*)addr_shared_memory + 2 *
         sizeof(int *);

    printf("Consumer%d get %d\n", value, **(addr_shared_memory + sizeof(int *)));
    (*(addr_shared_memory + sizeof(int *)))++;

    int sem_op_v = semop(semaphore, consumer_free, 2);
    if (sem_op_v == -1)
    {
        perror("Can't semop \n");
        exit(1);
    }
  }
}

// Производитель
void producer(int semaphore, int value)
{
  while(1)
  {
    sleep(2);
    int sem_op_p = semop(semaphore, producer_grab, 2);
    if (sem_op_p == -1)
    {
        perror("Can't semop \\n");
        exit(1);
    }

    if ((char*)(*addr_shared_memory) == ((char*)(addr_shared_memory) + 2 * sizeof
        (int *) + 5 * sizeof(int)))
      (*addr_shared_memory) = (char*)addr_shared_memory + 2 * sizeof(int *);

    *(*addr_shared_memory) = ((char*)(*addr_shared_memory) - (char*)
        addr_shared_memory) - 16;
    printf("Producer%d put %d\n", value, *(*addr_shared_memory));
    (*addr_shared_memory)++;

    int sem_op_v = semop(semaphore, producer_free, 2);
    if (sem_op_v == -1)
    {
        perror("Can't semop \n");
        exit(1);
    }
  }
}

int main()
{
  int process;
  int consumers[3];
  int producers[3];
  // Создание семафора
  semaphore = semget(IPC_PRIVATE, 3, IPC_CREAT | PERMS);
  int se = semctl(sem, EMPTYCOUNT, SETVAL, COUNT);
  int sf = semctl(sem, FULLCOUNT, SETVAL, 0);
  int sb = semctl(sem, BIN, SETVAL, 1);
  // Объявление разделяемого сегмента
  shared_memory = shmget(IPC_PRIVATE, 2 * sizeof(int *) + 5 * sizeof(int),
      IPC_CREAT | PERMS);
  addr_shared_memory = shmat(shared_memory, 0, 0);
  *(addr_shared_memory) = (char*)addr_shared_memory + 2 * sizeof(int *);
  *(addr_shared_memory + sizeof(int *)) = (char*)addr_shared_memory + 2 * sizeof(
      int *);

  // Создание процессов
  for (int i = 0; i < COUNT; i++) {
    if (-1 == (producers[i] = fork()))
    {
      return 1;
    }
    else if (0 == producers[i])
    {
      producer(semaphore, i);
      exit(0);
    }

    if (-1 == (consumers[i] = fork()))
    {
      return 1;
    }
    else if (0 == consumers[i])
    {
      consumer(semaphore, i);
      exit(0);
    }
  }

  signal(SIGINT, catch_sigp);
  int status;
  wait(&status);

  // Очистка памяти
  shmctl(shared_memory, IPC_RMID, NULL);
  semctl(semaphore, 0, IPC_RMID, 0);
  return 0;
}
```

Читатели-писатели

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <stdio.h>
#include <sys/shm.h>
#include <signal.h>

#define WRITER 3
#define READER 5

#define PERMS S_IRWXU | S_IRWXG | S_IRWXO

// количество читателей
#define READERCOUNT 0
// активные писатели
#define ACTIVEWRITER 1
// очередь писателей
#define WRITERCOUNT 2
// очередь читателей
#define READERWAIT 3

int semaphore;
int shared_memory;
int *addr_shared_memory;

// Массив структур
struct sembuf start_read[] = { {READERWAIT, 1, SEM_UNDO}, {ACTIVEWRITER, 0,
    SEM_UNDO}, {WRITERCOUNT, 0, SEM_UNDO}, {READERCOUNT, 1, SEM_UNDO}, {READERWAIT
    , -1, SEM_UNDO} };
struct sembuf stop_read[] = { {READERCOUNT, -1, SEM_UNDO} };
struct sembuf start_write[] = { {WRITERCOUNT, 1, SEM_UNDO}, {READERCOUNT, 0,
    SEM_UNDO}, {ACTIVEWRITER, -1, SEM_UNDO}, {WRITERCOUNT, -1, SEM_UNDO}, };
struct sembuf stop_write[] = { {ACTIVEWRITER, 1, SEM_UNDO} };

// собственный обработчик сигнала ctrl-c
void catch_sigp(int sig_numb)
{
    signal(sig_numb, catch_sigp);
    shmctl(shared_memory, IPC_RMID, NULL);
    semctl(semaphore, 0, IPC_RMID, 0);
}

void StartRead()
{
    semop(semaphore, start_read, 3);
}

void StopRead()
{
    semop(semaphore, stop_read, 1);
}

void StartWrite()
{
    semop(semaphore, start_write, 4);
}

void StopWrite()
{
    semop(semaphore, stop_write, 1);
}

// Читатель
void Reader(int value)
{
    while (1)
    {
            StartRead();
            printf("Reader%d get = %d\n", value, *addr_shared_memory);
            StopRead();
            sleep(1);
    }
}

// Писатель
void Writer(int value)
{
    while (1)
    {
        StartWrite();
        (*addr_shared_memory)++;
        printf("Writer%d put = %d\n", value, *addr_shared_memory);
        StopWrite();
        sleep(2);
    }
}

int main()
{
  // Создание семафора
  semaphore = semget(IPC_PRIVATE, 4, IPC_CREAT | PERMS);
  int sr = semctl(sem, READERCOUNT, SETVAL, 0);
  int sa = semctl(sem, ACTIVEWRITER, SETVAL, 1);
  int sw = semctl(sem, WRITERCOUNT, SETVAL, 0);

// Объявление разделяемого сегмента
shared_memory = (int*)shmget(IPC_PRIVATE, sizeof(int), IPC_CREAT | PERMS);
addr_shared_memory = shmat(shared_memory, 0, 0);

// Создание процессов
int processes[READER + WRITER];
int parent = getpid();

for (int i = 0; i < WRITER + READER; i++)
  {
      processes[i] = fork();
      if (getpid() != parent)
      {
          for (int j = 0; j < i; j++)
              processes[j] = -1;
          break;
      }
  }

  for (int i = 0; i < WRITER + READER; i++)
  {
      if (processes[i] != 0)
          continue;

      if (i < WRITER)
      {
    Writer(i);
          return 0;
      }
      else
      {
          Reader(i - WRITER);
          return 0;
      }
  }

signal(SIGINT, catch_sigp);
int status;
wait(&status);

// Очистка памяти
shmctl(shared_memory, IPC_RMID, NULL);
semctl(semaphore, 0, IPC_RMID, 0);
return 0;
}
```

Билет 18 (Пример из ЛР флаги, алгоритм Деккера, алгоритм Лампорт)

18.2 Защищенный режим: перевод компьютера в защищенный режим – реализация – пример кода из лабораторной работы.

```
.386p ; Разрешение трансляции всех, в том
            ; числе привилегированных команд МП 386
            ; и 486

descr struc  ; Структура для описания дескриптора сегмента
    limit   dw 0    ; Граница (биты 0..15)

    base_l  dw 0    ; База, биты 0..15
    base_m  db 0    ; База, биты 16..23
    attr_1  db 0    ; Байт атрибутов 1
    arrt_2  db 0    ; Граница(биты 16..19) и атрибуты 2
    base_h  db 0    ; База, биты 24..31
descr ends

data segment ; Начало сегмента данных
; Таблица глобальных дескрипторов GDT
    gdt_null descr <0,0,0,0,0,0> ; Нулевой дескриптор
    gdt_data descr <data_size-1,0,0,92h,0,0> ; Сегмент данных, селектор 8
    gdt_code descr <code_size-1,0,0,98h,0,0> ; Сегмент команд, селектор 16
    gdt_stack descr <255,0,0,92h,0,0> ; Сегмент стека, селектор 24
    gdt_screen descr <4095,8000h,0Bh,92h,0,0> ; Видеобуфер, селектор 32
    gdt_size=$-gdt_null ; Размер GDT
; Поля данных программы
    pdescr dq 0 ; Псевдодескриптор
    mes db "Real mode$" ; Сообщение для вывода в режиме реальном
    mes1 db "Protected mode$" ; Сообщение для вывода в режиме защищенном
    data_size=$-gdt_null ; Размер сегмента данных
data ends ; Конец сегмента данных

text segment 'code' use16 ; 16-разрядный режим
                    ; в первой части нам его достаточно
    assume CS:text, DS:data

main proc
        xor EAX,EAX ; Очистим EAX
        mov AX,data ; Загрузим в DS адрес сегмента данных
        mov DS,AX ; Адрес сегмента данных
; Вычислим 32-битовый линейный адрес сегмента данных и загрузим
; в дескриптор сегмента данных в GDT
        shl EAX,4 ; В EAX линейный базовый адрес
        mov EBP,EAX ; Сохраним его в EBP
        mov EBX,offset gdt_data ; В EBX адрес дескриптора
        mov [BX].base_l,AX ; Загрузим младшую часть базы
        rol EAX,16 ; Обмен старшей и младшей половин в EAX
        mov [BX].base_m,AL ; Загрузим среднюю чатсь базы
; Аналогично для сегмента команд
        xor EAX,EAX
        mov AX,CS
        shl EAX,4
        mov EBX,offset gdt_code
        mov [BX].base_l,AX
        rol EAX,16
        mov [BX].base_m,AL
; Аналогично для сегмента стека
        xor EAX,EAX
        mov AX, SS
        shl EAX,4
        mov EBX,offset gdt_stack
        mov [BX].base_l,AX
        rol EAX,16
        mov [BX].base_m,AL
; Подготовим псевдодескриптор pdescr и загрузим регистр GDTR
        mov dword ptr pdescr+2,EBP ; База GDT, биты 0..31
        mov word ptr pdescr,gdt_size-1 ; Граница GDT
        lgdt pdescr ; Загрузим регистр GDTR
; Подготовимся к переходу в защищенный режим
        cli ; Запрет аппаратных прерывания (маскируемых)
        mov AL,80h ;  Запрет NMI, немаскируемых прерываний
        out 70h,AL
; Переходим в защищенный режим
        mov EAX,CR0 ; Получим содержимое CR0
        or EAX,1 ; Установим бит PE
        mov CR0,EAX ; Запишем назад
; Теперь процессор работает в защищенном режиме
; Загружаем в CS:IP селектор:смещение точки continue
; и заодно очищаем очередь команд
        db 0EAh ; Код команды far jmp
        dw offset continue ; смещение
        dw 16 ; селектор сегмента команд

continue:
; Делаем адресуемыми данные
        mov AX,8 ; Селектор сегмента данных
        mov DS,AX
; Делаем адресуемым стек
        mov AX,24 ; Селектор сегмента стека
        mov SS,AX
; Делаем адресуемыми видеобуфер и выводим сообщение о переходе
        mov AX,32 ; Селектор сегмента видеобуфера
        mov ES,AX
        mov BX,800 ; Начальное смещение на экране
        mov CX,15 ; Число выводимых символов
        mov SI,0 ; Итератор (смещение от начала)

screen:
        mov EAX,word ptr mes1[SI] ; Символ для вывода со смещением SI
        mov ES:[BX],EAX ; Вывод в видеобуфер
        add BX,2 ; Смещаемся в видеобуфере
        inc SI ; Следующий символ строки
        loop screen ; Цикл вывода на экран
; Подготовим переход в реальный режи
; Сформируем и загрузим дескриптор для реального режима
; В нашем варианте не обязательно загружать FFFF, так как 16 битная система
        mov gdt_data.limit,0FFFFh ; Граница сегмента данных
        mov gdt_code.limit,0FFFFh ; Граница сегмента кода
        mov gdt_stack.limit,0FFFFh ; Граница сегмента стека
        mov gdt_screen.limit,0FFFFh ; Граница сегмента видеобуфера

        mov AX,8 ; Загрузим теневой регистр
        mov DS,AX ; сегмента данных
        mov AX,24 ; Загрузим теневой регистр
        mov SS,AX ; сегмента стека
        mov AX,32 ; Загрузим теневой регистр
        mov ES,AX ; сегмента видеобуфера
; Выполним дальний переход для того, чтобы заново
; загрузить слектор в регистр CS и модифицировать его теневой
        db 0EAh ; Командой дальнего перехода
        dw offset go ; загрузим теневой регситр
        dw 16 ; сегмента команд
; Переключим режим процессора
go:
        mov EAX,CR0 ; Получим содержимое CR0
        and EAX,0FFFFFFFEh ; Сбросим бит PE
        mov CR0,EAX ; Запишем назад
        db 0EAh ; Код команды far jmp
        dw offset return ; Смещение
        dw text ; Сегмент
; Теперь процессор снова работает в реальном режиме
; Восстановим операционную среду
return:
        mov AX,data ; Восстановим
        mov DS,AX ; адресуемость данных
        mov AX,stk ; Восстановим



        mov SS,AX ; адресуемость стека
; Разрешим аппартаные(маскируемые) и немаскируемые прерывания
        sti ; Разрешение аппаратных
        mov AL,0 ; Разрешение немаксируемых
        out 70h,AL
; Проверим выполнение функций DOS после возврата в реальный режим
        mov AH,09h ; Вывод сообщения
        mov EDX,offset mes
        int 21h
        mov AX,4C00h ; Завершение программы
        int 21h
        main endp

        code_size=$-main ; Размер сегмента кода(команд)
text ends

stk segment stack 'stack' ; Сегмент стека
        db 256 dup ('^')
stk ends

end main
```

Билет 21 (Пример из ЛР Запуск новой программы)

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  pid_t childpid1;
  pid_t childpid2;
  if ((childpid1 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди
     тельском процессе */
  {
    perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли
       бо */
    exit(1);                                          /* таблица заполнена) */
  }
  if (childpid1 == 0)
  { /* здесь располагается дочерний код */
    printf("Child1 forked \n");
    printf("Child1 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
       (), getpgrp());

    char msg[] = "exec() called from child1";

    /* обращаемся к программе echo, параметрами передаём её имя и текст для печат
       и */
    /* NULL - завершает список параметров */
        if (execlp("echo", "echo", msg, NULL) == -1) /* при успешном вызове exec
           () заменяет адресное пространство процесса */
        {
            perror("Can't exec"); /* - эти вызовы произойдут только в том случае,
                если exec() не выполнится*/
      exit(1);
        }
    return 0;

  }

  if ((childpid2 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди
     тельском процессе */
  {
    perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли
       бо */
    exit(1);                                          /* таблица заполнена) */
  }
  if (childpid2 == 0)
  { /* здесь располагается дочерний код */
    printf("Child2 forked \n");
    printf("Child2 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
       (), getpgrp());

    char msg[40] = "exec() called from child2";

    /* обращаемся к программе echo, параметрами передаём её имя и текст для печат
       и */
    /* NULL - завершает список параметров */
    if (execlp("echo", "echo", msg, NULL) == -1) /* при успешном вызове exec() за
       меняет адресное пространство процесса */
        {
            perror("Can't exec"); /* - эти вызовы произойдут только в том случае,
                if exec() не выполнится*/
            exit(1);
        }
  }

  if (childpid1 != 0 && childpid2 != 0)
  { /* здесь располагается родительский код */
    printf("Parent id = %d, child1 id = %d, child2 id = %d, group id = %d \n",
       getpid(), childpid1, childpid2, getpgrp());
    int status1;
    childpid1 = wait(&status1);
    printf("Child has finished: PID = %d\n", childpid1);

    int status2;
    childpid2 = wait(&status2);
    printf("Child has finished: PID = %d\n", childpid2);

    if (WIFEXITED(status1) && WIFEXITED(status2))
      printf("Children exited with code %d and %d\n", WEXITSTATUS(status1),
        WEXITSTATUS(status2));
    else
      printf("Child terminated abnormally\n");
  }
  return 0;
}
```

Билет 19 (Пример из ЛР Процессе зомби, wait, pipe Unix)

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  pid_t childpid1;
  pid_t childpid2;
  if ((childpid1 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди
     тельском процессе */
  {
    perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли
       бо */
    exit(1);                                          /* таблица заполнена) */
  }
  if (childpid1 == 0)
  { /* здесь располагается дочерний код */
    printf("Child1 forked \n");
    printf("Child1 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
       (), getpgrp());

    char msg[] = "exec() called from child1";

    /* обращаемся к программе echo, параметрами передаём её имя и текст для печат
       и */
    /* NULL - завершает список параметров */
        if (execlp("echo", "echo", msg, NULL) == -1) /* при успешном вызове exec
           () заменяет адресное пространство процесса */
        {


              perror("Can't exec"); /* - эти вызовы произойдут только в том случае,
                 если exec() не выполнится*/
        exit(1);
          }
    return 0;
  }

  if ((childpid2 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди
     тельском процессе */
  {
    perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли
       бо */
    exit(1);                                          /* таблица заполнена) */
  }
  if (childpid2 == 0)
  { /* здесь располагается дочерний код */
    printf("Child2 forked \n");
    printf("Child2 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
       (), getpgrp());

    char msg[40] = "exec() called from child2";

    /* обращаемся к программе echo, параметрами передаём её имя и текст для печат
       и */
    /* NULL - завершает список параметров */
    if (execlp("echo", "echo", msg, NULL) == -1) /* при успешном вызове exec() за
       меняет адресное пространство процесса */
        {
            perror("Can't exec"); /* - эти вызовы произойдут только в том случае,
                if exec() не выполнится*/
            exit(1);
        }
  }

  if (childpid1 != 0 && childpid2 != 0)
  { /* здесь располагается родительский код */
    printf("Parent id = %d, child1 id = %d, child2 id = %d, group id = %d \n",
       getpid(), childpid1, childpid2, getpgrp());
    int status1;
    childpid1 = wait(&status1);
    printf("Child has finished: PID = %d\n", childpid1);

    int status2;
    childpid2 = wait(&status2);
    printf("Child has finished: PID = %d\n", childpid2);

    if (WIFEXITED(status1) && WIFEXITED(status2))
      printf("Children exited with code %d and %d\n", WEXITSTATUS(status1),
         WEXITSTATUS(status2));
    else
      printf("Child terminated abnormally\n");
  }
  return 0;
}
```

Системный вызов wait(&status) вызывается в теле предка. Системный вызов wait() блокирует родительский процесс до момента завершения дочернего. При их завершении предок получает статус завершения потомка.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{

  pid_t childpid1;
  pid_t childpid2;
  if ((childpid1 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди
     тельском процессе */
  {
    perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли
       бо */
    exit(1);                                          /* таблица заполнена) */
  }
  if (childpid1 == 0)
  { /* здесь располагается дочерний код */
    printf("Child1 forked \n");
    printf("Child1 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
       (), getpgrp());
    getchar();
    printf("Child1 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
       (), getpgrp());
    return 0;
  }

  if ((childpid2 = fork()) == -1) /* если fork завершился успешно, pid > 0 в роди
     тельском процессе */
  {
    perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-ли
       бо */
    exit(1);                                          /* таблица заполнена) */
  }
```

```
  if (childpid2 == 0)
  { /* здесь располагается дочерний код */
    printf("Child2 forked \n");
    printf("Child2 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
       (), getpgrp());
    getchar();
    printf("Child2 id = %d, parent id = %d, group id = %d \n", getpid(), getppid
       (), getpgrp());
  }

  if (childpid1 != 0 && childpid2 != 0)
  { /* здесь располагается родительский код */
    printf("Parent id = %d, child1 id = %d, child2 id = %d, group id = %d \n",
       getpid(), childpid1, childpid2, getpgrp());
    int status1;
    childpid1 = wait(&status1);
    printf("Child has finished: PID = %d\n", childpid1);

    int status2;
    childpid2 = wait(&status2);
    printf("Child has finished: PID = %d\n", childpid2);

    if (WIFEXITED(status1) && WIFEXITED(status2))
      printf("Children exited with code %d and %d\n", WEXITSTATUS(status1),
         WEXITSTATUS(status2));
    else
      printf("Child terminated abnormally\n");
  }
  return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

#define ANSI_COLOR_GREEN "\x1b[32m"
#define ANSI_COLOR_RESET "\x1b[0m"

#define N 20

int main()
{
    int status;
    pid_t childpid1, childpid2;
    char buffer[N];
    char message1[N] = "hello";
    char message2[N] = "goodbye";
    int channel1[2], channel2[2];

    if (pipe(channel1) == -1)
    {
        perror("Can not pipe.\n");
        return 1;
    }

    childpid1 = fork();

    if (childpid1 == -1)
    {
        perror("Can not fork.\n");
        return 1;
    }
    else if (childpid1 == 0)
    {
        close(channel1[0]);
        write(channel1[1], message1, sizeof(message1));
        exit(0);
    }

    if (pipe(channel2) == -1)
    {
        perror("Can not pipe.\n");
        return 1;
    }

    childpid2 = fork();

    if (childpid2 == -1)
    {
        perror("Can't fork.\n");
        return 1;
    }
    else if (childpid2 == 0)
    {
        close(channel2[0]);
        write(channel2[1], message2, sizeof(message2));
        exit(0);
    }
    else
    {
        wait(&status);

            if (WIFEXITED(status))
            {
                printf(ANSI_COLOR_GREEN "child process exit success\n"
                   ANSI_COLOR_RESET);

                close(channel1[1]);
                read(channel1[0], buffer, sizeof(buffer));
                printf("message = %s\n", buffer);

                close(channel2[1]);
                read(channel2[0], buffer, sizeof(buffer));
                printf("message = %s\n", buffer);
            }
        }

        return 0;
    }
```

Билет 7 (Пример из ЛР код и заполнение дескрипторов GDT из лабораторной работы по защищенному режиму)

```
descr struc  ; Структура для описания дескриптора сегмента
    limit   dw 0    ; Граница (биты 0..15)
    base_l  dw 0    ; База, биты 0..15
    base_m  db 0    ; База, биты 16..23
    attr_1  db 0    ; Байт атрибутов 1
    arrt_2  db 0    ; Граница(биты 16..19) и атрибуты 2
    base_h  db 0    ; База, биты 24..31
descr ends

intr struc ; Структура для описания дескрипторов прерываний
    offs_l  dw 0    ; Смещение обработчика, нижняя часть (биты 0..15)
    sel     dw 0    ; Селектор сегмента команд
    rsrv    db 0    ; Зарезервировано
    attr    db 0    ; Атрибуты
    offs_h  dw 0    ; Смещение обработчика, верхняя часть (биты 16..31)
intr ends

GDT label   byte ; Таблица глобальных дескрипторов
    gdt_null descr <> ; Нулевой дескриптор
    gdt_flatDS descr <0FFFFh,0,0,92h,0CFh,0>
                                ; переключение в 32-х битную модель памяти flat
                                ; с лимитом в 4 Гб и страничной адресацией
    gdt_16bitCS descr <RM_seg_size-1,0,0,98h,0,0>
                            ; 16-битный 64-килобайтный сегмент кода с базой
                              RM_seg
    gdt_32bitCS descr <PM_seg_size-1,0,0,98h,0CFh,0>
                            ; 32-битный 4-гигабайтный сегмент кода с базой PM_seg

gdt_32bitDS descr <PM_seg_size-1,0,0,92h,0CFh,0>
                  ; 32-битный 4-гигабайтный сегмент данных с базой PM_seg
gdt_32bitSS descr <stack_l-1,0,0,92h,0CFh,0>
                ; 32-битный 4-гигабайтный сегмент данных с базой stack_seg
gdt_size = $-GDT ; размер нашей таблицы GDT+16байт (на саму метку)
; сегмент видеобуфера рассчитывается как смещение

gdtr dw gdt_size-1 ; Лимит GDT
    dd ? ; Линейный адрес GDT

; Имена для селекторов
SEL_flatDS equ 8
SEL_16bitCS equ 16
SEL_32bitCS equ 24
SEL_32bitDS equ 32
SEL_32bitSS equ 40

IDT label   byte ; Таблица дескрипторов прерываний IDT
trap1 intr 13 dup (<0, SEL_32bitCS,0, 8Fh, 0>)
                ; Первые 32 элемента таблицы (отведены под исключения)
trap13 intr <0, SEL_32bitCS,0, 8Fh, 0> ; Исключение общей защиты
trap2 intr 18 dup (<0, SEL_32bitCS,0, 8Fh, 0>)
                ; Первые 32 элемента таблицы (отведены под исключения)
int08 intr <0, SEL_32bitCS,0, 8Eh, 0> ; Дескриптор прерывания от таймера
int09 intr <0, SEL_32bitCS,0, 8Eh, 0> ; Дескриптор прерывания от клавиатуры
idt_size = $-IDT ; Размер IDT

idtr dw idt_size-1 ; Лимит IDT
    dd ? ; Линейный адрес IDT

idtr_real dw 3FFh,0,0 ; содержимое регистра IDTR в реальном режиме
```