Задачи коллоквиума 2016

1. В файловой системе используются битовые массивы для хранения информации о свободных и занятиях блоках. Область блоков содержит $N_{блоков}$. Написать функцию, принимающую в качестве параметров указатель на область памяти, в которой находится битовый массив и размер (в блоках) области блоков файловой системы, которая находит и выводит на стандартное устройство вывода размер самой большой непрерывной области свободных блоков файловой системы.

2. В файловой системе используются битовые массивы для хранения информации о свободных и занятиях блоках. Область блоков содержит $N_{блоков}$. Написать функцию, принимающую в качестве параметров указатель на область памяти, в которой находится битовый массив и размер (в блоках) области блоков файловой системы, которая находит и выводит на стандартное устройство вывода количество непрерывных областей свободных блоков файловой системы, каждая из которых имеет размер, превышающий размер предшествующей области занятых блоков. Считаем, что область блоков занимает $N_{блоков}$, и что начальные блоки области заняты.

   Считаем, что от начала области располагается массив битов. i-й бит содержит информацию о статусе i-го блока (занят/свободен – как конкретно кодируется, на выбор студента, проблему машинного представления не рассматриваем (т.е. не требуем какого-либо учета при программировании в каком порядке располагаются байты в переменных различных типов).

3. В компьютере используется 4-х сегментная модель организации памяти. Написать программу (функцию), реализующую преобразование виртуального сегментного адреса в физический (считаем, что используются все 4 сегмента). Ситуацию прерывания реализовать отправкой процессом сигнала SigUsr самому себе. Описать в программе все необходимые структуры данных, при учете того, что в машине используются виртуальные и физические адреса, равные по размеру usigned int.

   Описать таблицу сегментов. Это массив записей: <размер сегмента><база>.
   Описать структуру исполнительного адреса <номер сегмента><смещение>
   Прерывание происходит если База+СМЕЩЕНИЕ >(больше) размера сегмента.

4. Пусть дан 32-х разрядный компьютер, использующий страничную модель организации памяти. Размер страницы – 4096 байтов. Написать программу (функцию), моделирующую алгоритм преобразования виртуального адреса в физический с использованием инвертированной таблицы страниц. Описать в программе все необходимые структуры данных, при учете того, что в машине используются виртуальные и физические адреса, равные по размеру int. Считаем, что объем физической памяти, по объему, равен объему виртуального адресного пространства и что размер беззнакового целого 32 бита. Также считаем, что всевозможные виртуальные страницы процесса размещены в физической памяти
   Описываем инвертированную таблицу страниц: Массив структур <PID><№ Вирт. страницы>
   Описываем виртуальный исполнительный адрес: <PID><Виртуальный адрес>
   <Виртуальный адрес> есть: <№ Вирт. страницы> <Смещение> - в беззнаковом целом. Нужно: вырезать номер страницы из виртуального адреса, найти в таблице

совпадение PID и номер страницы – номер записи есть номер физической страницы. Совпадение всегда есть (все странички находятся в памяти).

5. Написать функцию, которая принимает в качестве параметра указатель на массив адресации блоков файла (из индексного дескриптора) которая выводит на стандартное устройство вывода найденный номер блока файла, имеющий минимальное значение. Размер элемента массива – int.

6. Написать функцию, которая принимает в качестве параметра указатель на массив адресации блоков файла (из индексного дескриптора) которая определяет и выводит на стандартное устройство размер файла в блоках. Размер элемента массива – int.

   В программе должен быть как-то определен размер блока в байтах или количество ссылок в одном блоке (здесь была некорректность в постановке задачи, мы ее откорректировали по ходу. Т.е. принимаем все и расчет количества по размеру блока в байтах и заданную константу). Все остальное просто. Перемещаемся по списку. Либо просматриваем весь до конца или доходим до первого NULL (или просто значения 0).

7. Что будет выведено на экран в результате работы фрагмента программы? Если возможны несколько вариантов – привести все. Предполагается, что обращение к функции вывода на экран прорабатывает атомарно и без буферизации. Все системные вызовы прорабатывают успешно. Подключение заголовочных файлов опущено.

```
msgId – идентификатор существующей пустой очереди сообщений.
struct
{
  long type;
  char data[1];
} msg;
……..
msg.type = 1; msg.data[0] = 'a'; msgsnd(msgId, &msg, 1, 0);
msg.type = 2; msg.data[0] = 'b'; msgsnd(msgId, &msg, 1, 0);
msg.type = 2; msg.data[0] = 'c'; msgsnd(msgId, &msg, 1, 0);
msg.type = 1; msg.data[0] = 'd'; msgsnd(msgId, &msg, 1, 0);
msgrcv(msgId, &msg, 1, 2, 0); putchar(msg.data[0]);
msgrcv(msgId, &msg, 1, 0, 0); putchar(msg.data[0]);
msgrcv(msgId, &msg, 1, 1, 0); putchar(msg.data[0]);
……
```

8. Что будет выведено на экран? Если возможны несколько вариантов – привести все. Предполагается, что обращение к функции вывода на экран прорабатывает атомарно и без буферизации. Все системные вызовы прорабатывают успешно. Подключение заголовочных файлов опущено.

```
int main()
{
      pid_t pid;
      int fd[2];
      int x = 3;
      pipe(fd);
      if( (pid = fork()) > 0 ) { read(fd[0], &x, sizeof(int));
                                 kill(pid, SIGKILL);
                                 wait(NULL);
```

```
      }
      else { printf("%d", x); x = 2; write(fd[1], &x, sizeof(int)); x = 1;
      }
      printf("%d", x);
      return 0;
}
```

О Т В Е Т Ы

1) Размер самой большой непрерывной области свободных блоков

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

/*  в freebits свободные блоки помечены '1', занятые '0' */
void print_size(const unsigned *freebits, int nblk)
{
    const unsigned ELEM_BIT = sizeof(*freebits) * CHAR_BIT;
    int startblk = -1;
    int maxfree = 0;
    int i;
    for (i = 0; i < nblk; ++i) {
        if (freebits[i / ELEM_BIT] & (1U << i % ELEM_BIT)) {
            if (startblk == -1) startblk = i;
        } else {
            if (startblk >= 0 && i - startblk > maxfree) {
                maxfree = i - startblk;
            }
            startblk = -1;
        }
    }
    if (startblk >= 0 && i - startblk > maxfree) {
        maxfree = i - startblk;
    }
    printf("%d\n", maxfree);
}
```

2) Количество областей свободных блоков

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

/*  в freebits свободные блоки помечены '1', занятые '0' */

void print_count(const unsigned *freebits, int nblk)
{
    const unsigned ELEM_BIT = sizeof(*freebits) * CHAR_BIT;
    int i;
    int prev = -1;
    int prevind = -1;
    int sizebusy = -1;
    int goodcount = 0;
    for (i = 0; i < nblk; ++i) {
        if (freebits[i / ELEM_BIT] & (1U << i % ELEM_BIT)) {
```

```
            // free block
            if (prev != 1) {
                // state change
                if (prevind >= 0) {
                    sizebusy = i - prevind;
                }
                prev = 1;
                prevind = i;
            }
        } else {
            // busy block
            if (prev != 0) {
                // state change
                if (prevind >= 0) {
                    if (i - prevind > sizebusy)
                        ++goodcount;
                }
                prev = 0;
                prevind = i;
            }
        }
    }
    if (prev == 1) {
        // state change
        if (prevind >= 0) {
            if (i - prevind > sizebusy)
                ++goodcount;
        }
    }
    printf("%d\n", goodcount);
}
```

3)

В 4-х сегментной модели памяти старшие два бита адреса отвечают за номер сегмента. Они используются для индексации таблицы сегментов. В ней хранится физический адрес сегмента и его размер. Проверяем, что смещение в сегменте меньше размера в сегменте, затем формируем и возвращаем физический адрес.

```
#include <signal.h>

struct Segment
{
    unsigned limit; /* размер сегмента */
    unsigned phys;  /* физический адрес */
};

struct Segment seg_table[4]; /* таблица сегментов */

unsigned translate(unsigned address)
{
    unsigned segno = address >> 30; /* получаем номер сегмента */
    unsigned offset = address & 0x3fffffff;
    if (offset >= seg_table[segno].limit) raise(SIGUSR1);
    return seg_table[segno].phys + offset;
}
```

4)

Каждая запись инвертированной таблицы страниц содержит PID процесса и номер виртуальной страницы или адрес виртуальной страницы). В таблице ищется запись с номером виртуальной страницы, совпадающим с номером виртуальной страницы входного адреса и с равным PID «контролирующего» процесса. Если нашли, индекс в таблице считается номером физической страницы, к нему приписывается смещение. Инвертированная таблица страниц должна содержать 2^20 записей (по одной для каждой физической страницы).

```
#include <signal.h>
#include <unistd.h>

struct IPTEntry
{
    pid_t pid;
    unsigned va; /* адрес виртуальной страницы, а не номер */
};

#define ENTRY_COUNT (1U << 20)

struct IPTEntry ipt[ENTRY_COUNT];

unsigned translate(unsigned address)
{
    unsigned offset = address & 0xFFF;
    unsigned va = address & ~0xFFF;
    pid_t pid = getpid();
    for (unsigned i = 0; i < ENTRY_COUNT; ++i) {
        if (ipt[i].pid == pid && ipt[i].va == va)
            return (i << 12) | offset;
    }
    raise(SIGUSR1);
    return 0;
}
```

5)

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

void read_block(int block_num, void *buf); /* считать номер блока
                                              block_num в буфер buf */

int scanblock1(int block_num, int block_size, int *p_min)
{
    int *buf = malloc(block_size);
    read_block(block_num, buf);
    int i;
    int limit = block_size / sizeof(int);
    for (i = 0; i < limit; ++i) {
        if (!buf[i]) return -1;
        if (buf[i] < *p_min) *p_min = buf[i];
    }
    free(buf);
    return 0;
}

int scanblock2(int block_num, int block_size, int *p_min)
```

```
{
    int *buf = malloc(block_size);
    read_block(block_num, buf);
    int i;
    int limit = block_size / sizeof(int);
    for (i = 0; i < limit; ++i) {
        if (!buf[i]) return -1;
        scanblock1(buf[i], block_size, p_min);
    }
    free(buf);
    return 0;
}

int scanblock3(int block_num, int block_size, int *p_min)
{
    int *buf = malloc(block_size);
    read_block(block_num, buf);
    int i;
    int limit = block_size / sizeof(int);
    for (i = 0; i < limit; ++i) {
        if (!buf[i]) return -1;
        scanblock2(buf[i], block_size, p_min);
    }
    free(buf);
    return 0;
}

void scanblocks(const int blocks[], int block_size)
{
    int min_block = 0;
    int i;
    /* 10 direct blocks  */
    for (i = 0; i < 10; ++i) {
        if (!blocks[i]) break;
        if (blocks[i] < min_block) min_block = blocks[i];
    }
    if (i == 10 && blocks[10]) {
        if (scanblock1(blocks[10], block_size, &min_block) >= 0 &&
blocks[11]) {
            if (scanblock2(blocks[11], block_size, &min_block) >= 0 &&
blocks[12]) {
                scanblock3(blocks[12], block_size, &min_block);
            }
        }
    }
    printf("%d\n", min_block);
}
```

6)

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

void read_block(int block_num, void *buf); /* считать номер блока
                                              block_num в буфер buf */

int scanblock1(int block_num, int block_size, int *p_count)
{
    int *buf = malloc(block_size);
    read_block(block_num, buf);
    int i;
    int limit = block_size / sizeof(int);
    for (i = 0; i < limit; ++i) {
        if (!buf[i]) return -1;
        ++(*p_count);
    }
    free(buf);
    return 0;
}

int scanblock2(int block_num, int block_size, int *p_min)
{
    int *buf = malloc(block_size);
    read_block(block_num, buf);
    int i;
    int limit = block_size / sizeof(int);
    for (i = 0; i < limit; ++i) {
        if (!buf[i]) return -1;
        scanblock1(buf[i], block_size, p_min);
    }
    free(buf);
    return 0;
}

int scanblock3(int block_num, int block_size, int *p_min)
{
    int *buf = malloc(block_size);
    read_block(block_num, buf);
    int i;
    int limit = block_size / sizeof(int);
    for (i = 0; i < limit; ++i) {
        if (!buf[i]) return -1;
        scanblock2(buf[i], block_size, p_min);
    }
    free(buf);
    return 0;
}
void scanblocks(const int blocks[], int block_size)
{
    int block_count = 0;
    int i;
    /* 10 direct blocks  */
    for (i = 0; i < 10; ++i) {
        if (!blocks[i]) break;
        ++block_count;
    }
    if (i == 10 && blocks[10]) {
        if (scanblock1(blocks[10], block_size, &block_count) >= 0
             && blocks[11]
```

```
            ){
               if (scanblock2(blocks[11], block_size, &block_count) >= 0
                    && blocks[12]
                   ){
                     scanblock3(blocks[12], block_size, &block_count);
               }
        }
    }
    printf("%d\n", block_count);
}
```

7) bad

8) 32 или 312