СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ

Компьютер — это проргаммно-управляемое устройство. Часть времени работой компьютера управляет ОС, другую часть — программа.

Принцип хранимой программы — процессор может выполнять только программу, находящуюся в оперативной памяти.

Мультипрограммность — в памяти хранится сразу несколько программ и процессор может быстро переключаться с одной на другую.

Мультизадачность — поддержка загрузки в память нескольких программ.

Терминал — совокупность клавиатуры и монитора; внешнее устройство; рабочее место программиста.

Квант — интервал процессорного времени, которое выделено конкретной задаче; главная задача квантования процессорного времени — обеспечить гарантированное время ответа.

Файловая система — набор правил, определяющих способ организации, хранения и именования данных на носителях информации.

Вторичная память — энергонезависимая память, предназначенная для долговременного хранения информации (её главная задача).

Файл — поименованная совокупность данных, может быть бессмысленная.

Каталог UNIX — файл в файловой системе. Содержимое каталога обрабатывается самой операционной системой, а не пользовательской программой. Каталог содержит файлы, которые в свою очередь могут быть каталогами — UNIX имеет иерархическую файловую систему.

Ядро — процессор с полноценным набором регистров (но имеется некоторая виртуализация, так как сейчас процессоры являются программно-управляем

Процесс — программа в стадии выполнения (программа, которая просто лежит на диске — это файл).

Планирование — постановка процессов в очередь (очев., что процессорное время получит процесс, находящийся первым в очереди) по каким-то принципам.

Планировщик — программа, которая отвечает за порядок получения процессорного времени процессами.

Диспечеризация — выделение процессорного времени.

Шина — медный провод. Если зажать в кулаке 16 проводов, это будет 16-разрядная шина. Это предполагает параллельную передачу данных. По каждой шине передаётся один разряд. В современных компьютерах, работая на определённых частотах, параллельная передача невозможна, только последовательная.

Регистры — неотъемлемая часть процессора (не память, процессор собственной памяти не имеет).

Порт — адрес, по которому процессор обращается к внешним устройствам.

Аппаратные прерывания — абсолютно асинхронные события в системе, так как не зависят от любой работы, выполняемой в системе.

1 Другое

1.1 Типы ОС

Однопрограммные пакетные обработчики — в памяти хранится одна программа, но она считана из пакета.

Мультипрограммные пакетные обработкичи — в памяти хранится несколько программ, считанных из пакета. Выполняются программы от начала и до конца — неопределённое время ожидания.

Системы разделения времени — процессорное время квантуется (очев., что системы разделения времени мультизадачные).

Наши компьютеры мультипрограммные и мультизадачные (одновременно выполняется несколько процессов; в однопроцессорных системах достигается засчёт быстрого переключения между задачами; в мультипроцессорных — задачи фактически могут выполняться параллельно на разных процессорах).

1.2 Прерывания

Операциями ввода-вывода управляет устройство; процессор от этого освобождён. Появилась необходимость информаировать процессор о завершении операций ввода-вывода — возникла система пререываний:

1) системные вызовы (программные прерывания),

2) исключения,

3) аппаратные прерывания:

   — прерывания от системного таймера (единственное периодическое),

   — прерывания от внешних устройств (информируют процессор о завершении операций ввода-вывода),

   — прерывания от действий оператора (KeyboardInterrupt, ...).

Процессор и память связаны локальной шиной, называемой локальной шиной памяти.

Процессор памяти не имеет и постоянно обменивается данными с оперативной памятью. Регистры — неотъемлемая часть процессора, это не память.

Процессор — регистры, управляемые передачей данных. В процессоре тоже есть микропрограммное управление. Процессор — тоже программно управляемое устройство.

Принцип хранимой программы (основной принцип фон Неймана) — процессор может выполнять программу, которая хранится в ОЗУ; и данные, и команды хранятся в одной и той же память; обращение к командам и данным выполняется только по адресу (расположены в памяти в последовательных адресах).

В состав процессора входит PC (IP в Intel) — всегда содержит адреса команд, которые надо считать из памяти (адрес следующей команды).

Настраивается на начальный адрес сначала, считывает следующую команду, увеличивает IP на размер следующей команды — базовый принцип работы вычислительных машин.

Внешние устройства адресуются также, как ячейки памяти. В компьютере всё адресуется. К внешним устройствам процессор обращается по адресу, но такой адрес называется портом. Порт — это адрес.

Если процессору надо обратиться к внешнему устройству, он устанавливает порт; дальше в зависимости от ввода/вывода; в зависимости от устройства, либо устанавливает, либо получает данные из шины данных.

В 3-м поколении ЭВМ была реализована идея распараллеливания функций. Функции управления внешними устройствами от процессора были переданы следующим устройствам:

— селекторы и мультиплексоры (в канальной архитектуре),

— контроллер (в шинной архитектуре); если в составе внешнего устройства — контроллер, если в составе материнской платы — адаптер.

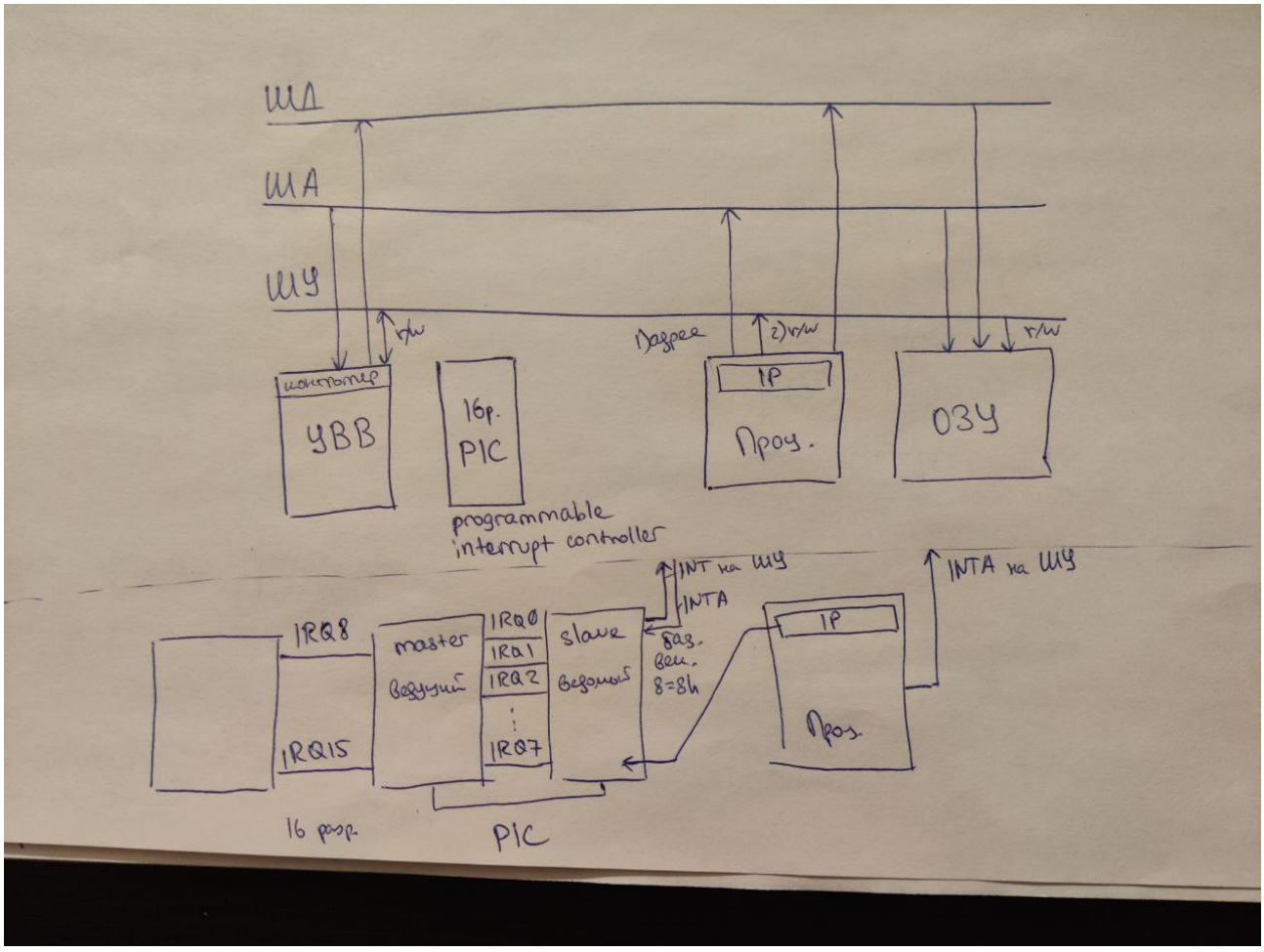
Контроллер по шине данных от процессора получает команды, которые управляют операциями ввода-вывода — появилась система прерываний.

Существует три типа прерываний (что бы Microsoft ни говорил):

1) Системные вызовы (вызов системы) — так называемые программные прерывания — программа прерывается, чтобы перейти на управление системного вызова — кода ядра.

2) Исключения — ошибки системы, связанные с разными событиями в системе; ZeroDivisionError — самая базовая. Разрядность в компах ограничена, нельзя представить ∞, поэтому ZeroDivisionError.

3) Аппаратные прерывания (*Interrupt) — аппаратное прерывание возникает в системе в результате завершения выполнения внешними устройствами операций ввода-вывода. Управляет не процессор, а контроллер или каналы — необходимо проинформировать процессор о том, что операция ввода-вывода завершена. То есть, аппаратные прерывания формируются контроллером прерываний, когда внешнее устройство завершает операции ввода-вывода.



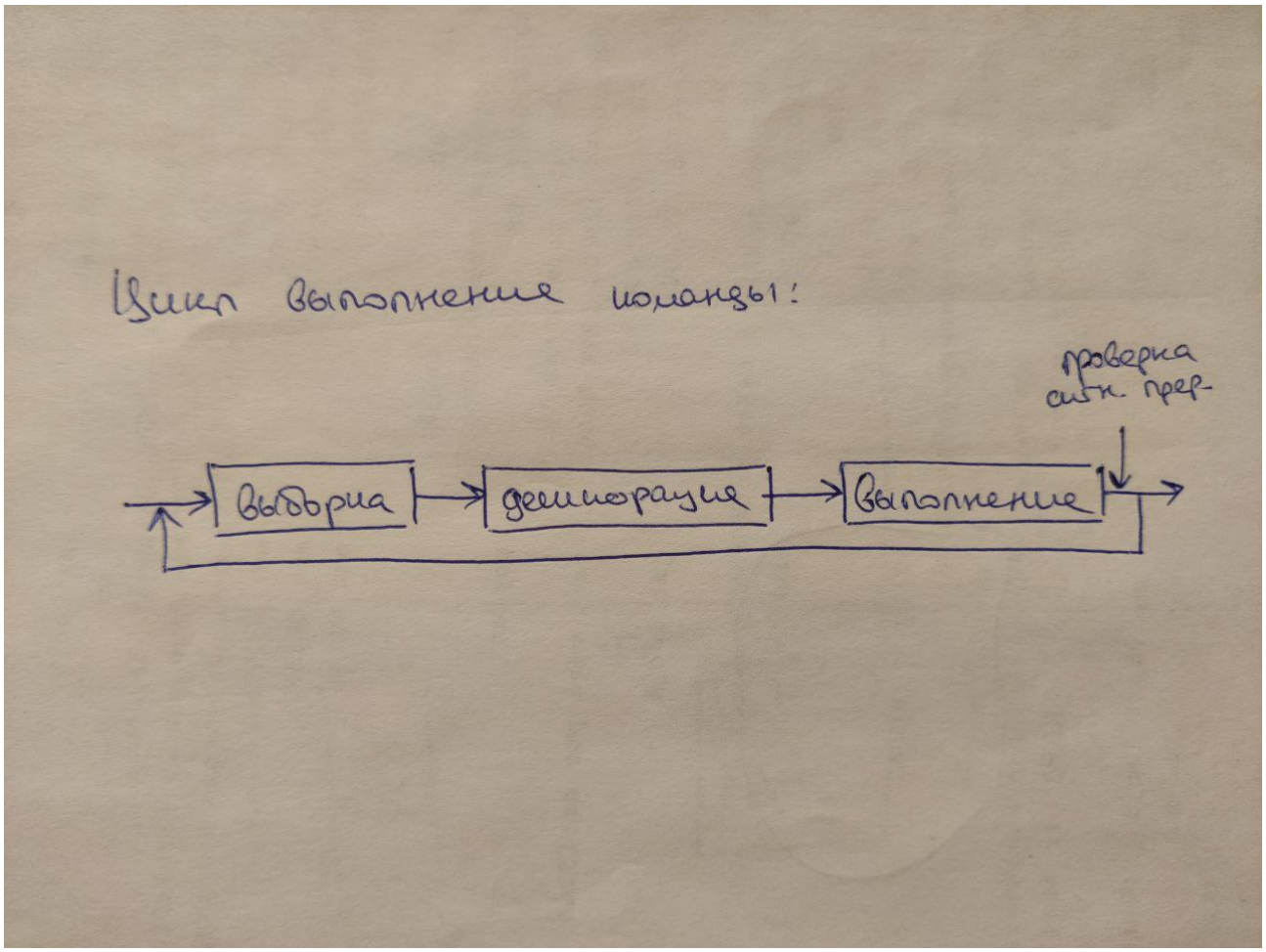
На линию поступает сигнал прерывания от контроллера внешних устройств.

IRQ0 — туда приходит прерывание от системного таймера, так называемый «тик». В реальном режиме формируется 18.2 раза в секунду. Единственное периодическое прерывание в системе. Все остальные прерывания от

внешних устройств возникают в случайным момент времени. Даже используются для генерации случайных чисел.

IRQ1 — туда приходят прерывания клавиатуры (разъём ПС/2); клавиатура ноутбука — IRQ1, мышь — IRQ12.

Когда на линию IRQ приходит сигнал прерывания от внешнего устройства; если этот сигнал прерывания замаскирован (разрешён), контроллер выставляет на шину управления сигнал INT.



Процессор проверяет наличие сигнала прерывания на выделенной ножке. Если сигнал прерывания пришёл, то процессор отсылает ответный сигнал — INTA. Потом выставляет на вектор прерываний (вектор прерываний — базовый вектор + номер линии IRQ). В реальном режиме (16 р.) базовый вектор контроллера равен 8. 8 + 0 = 8; отсюда INT 8h название.

Вектор прерываний (ВП) по шине данных поступает в процессор. Процессор по полученному вектору в реальном режиме обращается к таблице векторов прерываний (расположена в первом 1 КБ, начиная с 0).

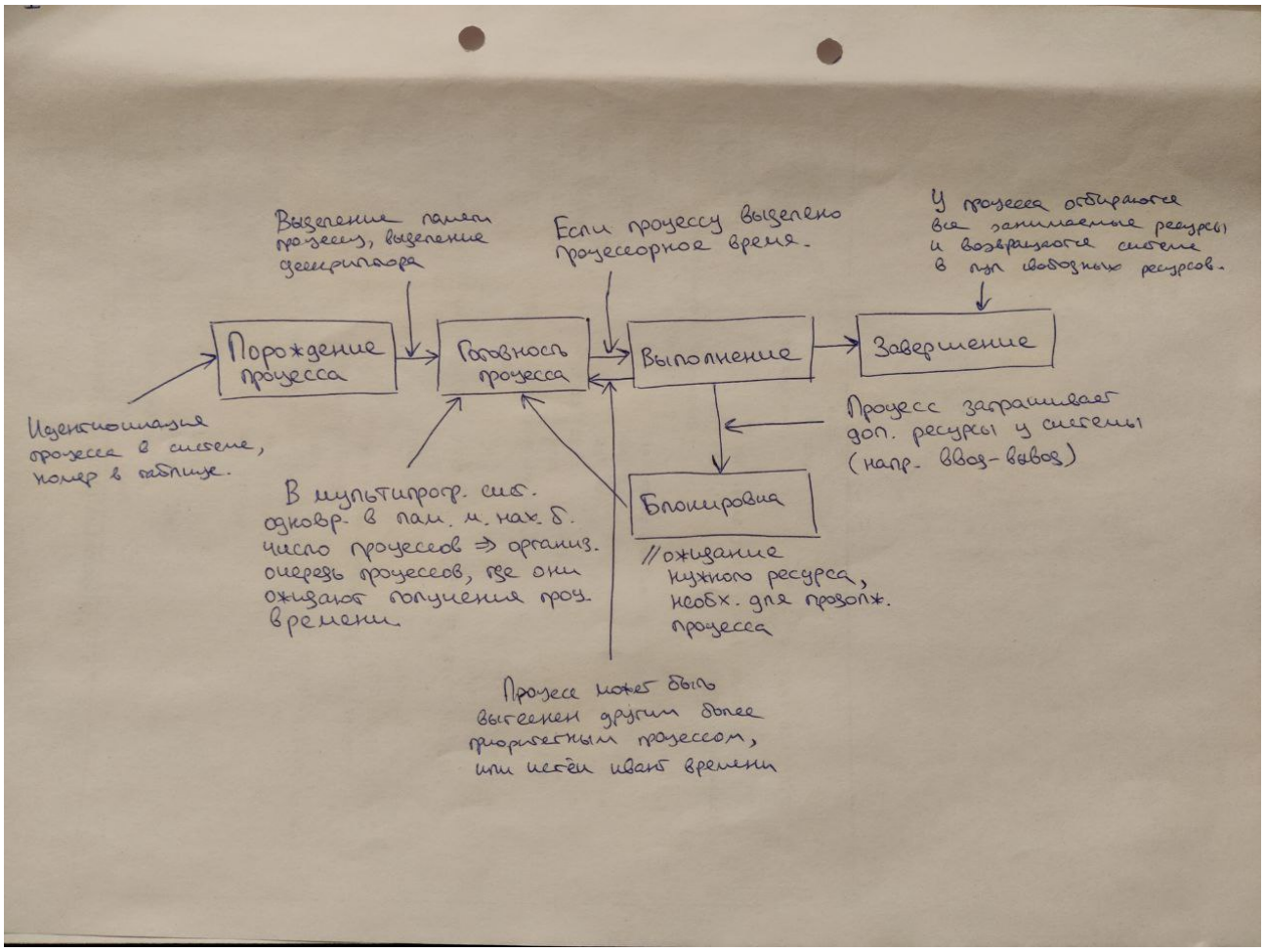
Сам контроллер прерываний тоже адресуется, тоже через порт и по

шине данных на контроллер посылается маска прерывания.

Тик приходит на линию IRQ0 контроллера прерываний и приводит к вызову обработчика прерывания от системного таймера.

1.3 Диаграмма состояний процесса

1.3.1 Стадии



1) Идентификация процесса. Ни одна программа не может работать с неопределёнными значениями; для этого задаём имя и типы переменных — это и есть идентификация переменных. Система также не может работать непонятно с чем — должна идентифицировать процесс. В системе выполняется большое число процессов — существуют средства описания процессов. В современном программировании этими средствами являются структуры (структуры появились в C).

2) Описание дескриптора процесса (дескриптор — описатель, структура,

предназначенная для описания процесса). Существует поле, отражающее текущее состояние процесса; в этой структуре находится указатель на другую структуру, в частности, на структуру, предназначенную для управления памятью. Процессор может выполнять только программы, находящиеся в памяти.

3) Система выделяет процессу память. Дескриптор должен содержать информацию о выделенной процессу памяти — это означает, что программа загружена в физическую память; псоле этого программа может начать выполняться.

Планирование — постановка процессов в очередь (очев., что процессорное время получит процесс, находящийся первым в очереди) по каким-то принципам.

Диспечеризация — выделение процессорного времени.

2 Взаимоисключение

Для того, чтобы не происходило race conditions, к разделяемым переменным (в общем случае, к разделяемым ресурсам) должен обеспечиваться монопольный доступ — если один процесс обратился к разделяемой переменной, никакой другой процесс к ней обратиться не может, пока тот процесс не освободит переменную. Монопольный доступ обеспечивается средствами взаимоисключения.

3 Семафоры

4 Коды из лекций

4.1 Взаимоисключение на флагах 1 (проблема)

```
1  prog_example1
2
3  fp1, fp2: logical
4
5  p1:
6       while (1):
7            while (fp2);
8            fp1 = 1;
9            CR1;
10           fp1 = 0;
11           PR1; // ???
12 end p1;
13
14 p2:
15      while (1):
16           while (fp1);
17           fp2 = 1;
18           CR2;
19           fp2 = 0;
20           PR2; // ???
21 end p2;
22
23 begin
24      fp1 = 0; fp2 = 0;
25      par begin
26           p1; p2;
27      par end;
28 end.
```

4.2 Взаимоисключение на флагах 2 (проблема)

```
1  prog_example2
2
3  fp1, fp2: logical
4
5  p1:
6       while (1):
7            fp1 = 1;
8            while (fp2);
9            CR1;
10           fp1 = 0;
11           PR1; // ???
12 end p1;
13
14 p2:
15      while (1):
16           fp2 = 1;
17           while (fp1);
18           CR2;
19           fp2 = 0;
20           PR2; // ???
21 end p2;
22
23 begin
24      fp1 = 0; fp2 = 0;
25      par begin
26           p1; p2;
27      par end;
28 end.
```

4.3 Взаимоисключение на флагах (решение Деккера)

```
fp1, fp2: logical;
que: int;

p1:
    while (1):
        fp1 = 1;
        while (fp2)
        {
            if (que == 2) then
            {
                fp1 = 0;
                while (que == 2);
                fp1 = 1;
            }
        }
    CR1;
    fp1 = 0;
    que = 2;
    PR1;
end p1;

p2:
    while (1):
        fp2 = 1;
        while (fp1)
        {
            if (que == 1) then
            {
                fp2 = 0;
                while (que == 1);
                fp2 = 1;
            }
        }
    CR2;
    fp2 = 0;
    que = 1;
    PR2;
end p2;
```

```
39
40 begin
41      fp1 = 0; fp2 = 0;
42      que = 1;
43      par begin
44          p1; p2;
45      par end;
46 end.
```

4.4 test_and_set

```
1  program use_test_and_set;
2  flag, c1, c2: logical;
3
4  p1:
5       while (1)
6       {
7            c1 = 1;
8            while (c1 == 1)
9                 test_and_set(c1, flag);
10           CR1;
11           flag = 0;
12           PR1;
13      }
14
15 p2:
16      while (1)
17      {
18           c2 = 1;
19           while (c2 == 1)
20                test_and_set(c2, flag);
21           CR2;
22           flag = 0;
23           PR2;
24      }
25
26 main()
27 {
28      flag = 0;
29      parbegin
30           p1; p2;
31      parend;
32 }
```

4.5 spin_lock, spin_unlock

```
1  void spin_unlock(spin_lock_t *c)
2  {
3      // c - conditional
4      *c = 0;
5  }
6
7  void spin_lock(spin_lock_t *c)
8  {
9      while (test_and_set(*c) != 0)
10         /* ресурс занят */
11 }
12
13 void spin_lock(spin_lock_t *c)
14 {
15     while (test_and_set(*c) != 0)
16         while (*c != 0);
17 }
```

4.6 Производство-потребление

```
1  se, sf, sb: semaphore;
2
3  producer:
4      while (1)
5      {
6          p(se);
7          p(sb);
8          N = N + 1;
9          v(sb);
10         v(sf);
11     }
12
13 consumer:
14     while (1)
15     {
16         p(sf);
17         p(sb);
18         N = N - 1;
19         v(sb);
20         v(se);
21     }
22
23 begin
24     se = N; sf = 0; sb = 1;
25     parbegin
26         producer; consumer;
27     parend;
28 end.
```

4.7 Алгоритм Лампорта — Булочная

```
1  var choosing: shared array[0..n-1] of boolean;
2  var number: shared array[0..n-1] of integer;
3  ...
4  repeat
5      choosing[i] := true;
6      number[i] = max(number[0], number[1],
7              ..., number[n-1]) + 1;
8      choosing[i] = false;
9      for j := 0 to n-1 do begin
10         while choosing[i] do (*nothing*);
11         while number[j] <> 0 and
12             (number[j], j) < (number[i], i)
13             do (*nothing*);
14     end;
15     (*critical section*)
16     number[i] := 0;
17     (*remainder section*)
18 until false;
```

4.8 Обедающие философы

```
1  var
2      forks: array[1..5] of semaphore;
3      i: integer;
4
5  begin
6      repeat
7          forks[i] := 1;
8          i := i - 1;
9      until i = 0;
10
11 parbegin
12     1: begin
13         left := 1; right := 2;
14     end;
15     2: begin
16         left := 2; right := 3
17     end;
18     ...
19     5: begin
20         left := 5; right := 1
21         repeat
22             // размышляет
23             p(forks[left], forks[right]); // освобождена
24             // ест
25             v(forks[left], forks[right]); // захват
26         forever;
27     end; // 5
28 parend;
29
30 end.
```

4.9 Простой монитор

```
1  monitor: resource;
2  var
3      busy: logical;
4      x: conditional;
5
6  procedure acquire;
7      begin
8          if busy then wait(x);
9          busy := true;
10     end;
11
12 procedure release;
13     begin
14         busy := false;
15         signal(x);
16     end;
17
18 begin
19     busy := false;
20 end.
```

4.10 Монитор «Кольцевой буфер» (производство-потребление)

```
1  monitor: resource;
2  var
3      bcircle: array[0..n-1] of type;
4      pos: 0..n; // текущая позиция
5      j: 0..n; // заполняемая позиция
6      k: 0..n; // освобождамая позиция
7      buffer_full, buffer_empty: conditional;
8
9  procedure producer(var data: type);
10 begin
11     if pos = n then
12         wait(buffer_empty);
13     bcircle[j] := data;
14     pos := pos + 1;
15     j := (j + 1) mod n;
16     signal(buffer_full);
17 end;
18
19 procedure consumer(var data: type);
20 begin
21     if (pos = 0) then
22         wait(buffer_full);
23     data := bcircle[k];
24     pos := pos - 1;
25     k := (k + 1) mod n;
26     signal(buffer_empty);
27 end;
28
29 begin
30     pos := 0;
31     j := 0;
32     k := 0;
33 end.
```

4.11 Монитор «Читатели-писатели»

```
1  monitor: resource;
2  var
3      nr: integer; // количество читателей
4      wrt: logical; // активный писатель
5      c_read, c_write: conditional; // can_read, can_write
6
7  procedure startread;
8  begin
9      if wrt or turn(c_write)
10         then wait(c_read);
11     nr := nr + 1;
12     signal(c_read);
13 end;
14 procedure stopread;
15 begin
16     nr := nr - 1;
17     if nr = 0 then
18         signal(c_write);
19 end;
20
21 procedure startwrite;
22 begin
23     if nr > 0 or wrt then
24         wait(c_write);
25     wrt := true;
26 end;
27 procedure stopwrite;
28 begin
29     wrt := false;
30     if turn(c_read) then
31         signal(c_read);
32     else
33         signal(c_write);
34 end;
35
36 begin
37     nr := 0;
38     wrt := false;
39 end.
```

5 Коды лабораторных

5.1 Производство-потребление

```
#define BUFSIZE 128
#define PERMS S_IRWXU | S_IRWXG | S_IRWXO
#define NP   3
#define NC   3
#define SE   0
#define SF   1
#define SB   2

struct sembuf start_produce[2] = { {SE, -1, 0}, {SB, -1, 0} };
struct sembuf stop_produce[2]  = { {SB,  1, 0}, {SF,  1, 0} };
struct sembuf start_consume[2] = { {SF, -1, 0}, {SB, -1, 0} };
struct sembuf stop_consume[2]  = { {SB,  1, 0}, {SE,  1, 0} };

int semid;
char *addr;
char **ptr_prod;
char **ptr_cons;
char *conv; // Conveyor
char *ch;

int flag = 1;
void sig_handler(int sig_num)
{
    flag = 0;
    printf("pid: %d, signal: %d\n", getpid(), sig_num);
}

void producer(const int semid)
{
    srand(time(NULL) + getpid());
    int exit_flag = 0;
    while (flag)
    {
        usleep((double)rand() / RAND_MAX * 1000000);
        int p = semop(semid, start_produce, 2);
        if (p == -1) { perror("p semop error p\n"); exit(1); }
```

```
37          if (*ch > 'z')
38          {
39              printf("Producer %d is about to exit\n", getpid());
40              exit_flag = 1;
41          }
42          else
43          {
44              **ptr_prod = *ch;
45              printf("Producer %d >>> %c (%p)\n", getpid(),
                    **ptr_prod, *ptr_prod);
46              (*ptr_prod)++;
47              (*ch)++;
48          }
49          int v = semop(semid, stop_produce, 2);
50          if (v == -1) { perror("p semop error v\n"); exit(1); }
51          if (exit_flag) { printf("Producer %d has exited with code
                0\n", getpid()); exit(0); }
52      }
53      exit(0);
54  }
55
56  void consumer(const int semid)
57  {
58      srand(time(NULL) + getpid());
59      int exit_flag = 0;
60      while (flag)
61      {
62          usleep((double)rand() / RAND_MAX * 1000000);
63          int p = semop(semid, start_consume, 2);
64          if (p == -1) { perror("c semop error p\n"); exit(1); }
65          printf("Consumer %d <<< %c (%p)\n", getpid(), **ptr_cons,
                *ptr_cons);
66          if (**ptr_cons == 'z')
67          {
68              printf("Consumer %d is about to exit\n", getpid());
69              exit_flag = 1;
70          }
71          else
72          {
73              (*ptr_cons)++;
74          }
```

```
75          int v = semop(semid, stop_consume, 2);
76          if (v == -1) { perror("c semop error v\n"); exit(1); }
77          if (exit_flag) { printf("Consumer %d has exited with code
               0\n", getpid()); exit(0); }
78      }
79      exit(0);
80  }
81
82  int main()
83  {
84      signal(SIGINT, sig_handler);
85
86      pid_t pids[NP + NC];
87
88      int memkey = 0;
89      int fd = shmget(memkey, BUFSIZE, IPC_CREAT | PERMS);
90      if (fd == -1) { perror("shmget\n"); exit(1); }
91
92      addr = (char*)shmat(fd, 0, 0);
93      if (addr == (char*)-1) { perror("shmat\n"); exit(1); }
94      ptr_prod = (char**)addr;
95      ptr_cons = (char**)((char*)ptr_prod + sizeof(char*));
96      ch = (char*)ptr_cons + sizeof(char*);
97      *ch = 'a';
98      conv = ch + sizeof(char);
99      *ptr_prod = conv;
100     *ptr_cons = conv;
101
102     int semkey = ftok("keyfile", 0);
103     if ((semid = semget(semkey, 3, IPC_CREAT | PERMS)) == -1) {
           perror("semget\n"); exit(1); }
104     int cse = semctl(semid, SE, SETVAL, BUFSIZE);
105     int csf = semctl(semid, SF, SETVAL, 0);
106     int csb = semctl(semid, SB, SETVAL, 1);
107     if (cse == -1 || csf == -1 || csb == -1) {
           perror("semctl\n"); exit(1); }
108
109     pid_t pid = -1;
110     for (int i = 0; i < NP; i++)
111     {
112         pid = fork();
```

```
113             if (pid == -1) { perror("p can't fork\n"); exit(1); }
114             if (pid == 0) { producer(semid); }
115             else { pids[i] = pid; }
116         }
117         for (int i = 0; i < NC; i++)
118         {
119             pid = fork();
120             if (pid == -1) { perror("c can't fork\n"); exit(1); }
121             if (pid == 0) { consumer(semid); }
122             else { pids[NP + i] = pid; }
123         }
124 
125         for (int i = 0; i < (NP + NC); i++)
126         {
127             check_macros(pids[i]);
128         }
129 
130         if (shmdt(addr) == -1) { perror("shmdt\n"); exit(1); }
131 
132         if (shmctl(fd, IPC_RMID, (void*)addr) < 0)
133         {
134             perror("rm shm error\n");
135             exit(1);
136         }
137 
138         if (semctl(semid, 0, IPC_RMID) < 0)
139         {
140             perror("rm sem error\n");
141             exit(1);
142         }
143 }
```

5.2 Читатели-писатели

```
1  #define SHMSIZE sizeof(int)
2  #define PERMS (S_IRWXU | S_IRWXG | S_IRWXO)
3  #define NW  3
4  #define NR  5
5
6  #define R_DELAY 1
7  #define W_DELAY 1
8
9  #define C_WAITING_R 0
10 #define C_ACTIVE_R 1
11 #define C_WAITING_W 2
12 #define B_ACTIVE_W 3
13
14 struct sembuf sem_start_read[] = {
15      { C_WAITING_R,   1, 0 },
16      { B_ACTIVE_W,    0, 0 },
17      { C_WAITING_W,   0, 0 },
18      { C_WAITING_R,  -1, 0 },
19      { C_ACTIVE_R,    1, 0 },
20 };
21
22 struct sembuf sem_stop_read[] = {
23      { C_ACTIVE_R,   -1, 0 },
24 };
25
26 struct sembuf sem_start_write[] = {
27      { C_WAITING_W,   1, 0 },
28      { C_ACTIVE_R,    0, 0 },
29      { B_ACTIVE_W,   -1, 0 },
30      { C_WAITING_W,  -1, 0 },
31 };
32
33 struct sembuf sem_stop_write[] = {
34      { B_ACTIVE_W,    1, 0 },
35 };
36
37 int semid;
38 int flag = 1;
39
```

```
40 void sig_handler(int sig_num)
41 {
42     flag = 0;
43     printf("pid: %d, signal catch: %d\n", getpid(), sig_num);
44 }
45
46 int start_read(const int semid)
47 {
48     return semop(semid, sem_start_read, 5);
49 }
50
51 int stop_read(const int semid)
52 {
53     return semop(semid, sem_stop_read, 1);
54 }
55
56 int start_write(const int semid)
57 {
58     return semop(semid, sem_start_write, 4);
59 }
60
61 int stop_write(const int semid)
62 {
63     return semop(semid, sem_stop_write, 1);
64 }
65
66 void reader(const int semid, const char *shm)
67 {
68     srand(getpid());
69     printf("R[%5d] created.\n", getpid());
70     while (flag)
71     {
72         usleep((double)rand() / RAND_MAX * 1000000 * R_DELAY);
73         if (start_read(semid) == -1)
74         {
75             perror("start read error\n");
76             exit(1);
77         }
78         // Critical section begin
79         printf("R[%5d]: %3d\n", getpid(), *((int*)shm));
80         // Critical section end
```

```
81          if (stop_read(semid) == -1)
82          {
83              perror("stop read error\n");
84              exit(1);
85          }
86      }
87      exit(0);
88  }
89
90  void writer(const int semid, char *shm)
91  {
92      srand(getpid());
93      printf("W[%5d] created.\n", getpid());
94      while (flag)
95      {
96          usleep((double)rand() / RAND_MAX * 1000000 * W_DELAY);
97          if (start_write(semid) == -1)
98          {
99              perror("start write error\n");
100             exit(1);
101         }
102         // Critical section begin
103         ++(*(int*)shm);
104         printf("W[%5d]: %3d\n", getpid(), *((int*)shm));
105         // Critical section end
106         if (stop_write(semid) == -1)
107         {
108             perror("stop write error\n");
109             exit(1);
110         }
111     }
112     exit(0);
113 }
114
115 int main()
116 {
117     signal(SIGINT, sig_handler);
118
119     pid_t pids[NW + NR];
120
121     int memkey = 0;
```

```
122     int shmid = shmget(memkey, SHMSIZE, IPC_CREAT | PERMS);
123     if (shmid == -1) { perror("shmget\n"); exit(1); }
124
125     char *shmaddr = (char*)shmat(shmid, NULL, 0);
126     if (shmaddr == (char*)-1) { perror("shmat\n"); exit(1); }
127
128     memset(shmaddr, 0, SHMSIZE);
129
130     int semkey = ftok("keyfile", 0);
131     if ((semid = semget(semkey, 4, IPC_CREAT | PERMS)) == -1)
132     {
133         perror("semget\n");
134         exit(1);
135     }
136
137     int cbsaw = semctl(semid, B_ACTIVE_W, SETVAL, 1);
138
139     if (cbsaw == -1)
140     {
141         perror("semctl\n");
142         exit(1);
143     }
144
145     pid_t pid = -1;
146     for (int i = 0; i < NW; i++)
147     {
148         pid = fork();
149         if (pid == -1) { perror("w can't fork\n"); exit(1); }
150         if (pid == 0) { writer(semid, shmaddr); }
151         else { pids[i] = pid; }
152     }
153     for (int i = 0; i < NR; i++)
154     {
155         pid = fork();
156         if (pid == -1) { perror("r can't fork\n"); exit(1); }
157         if (pid == 0) { reader(semid, shmaddr); }
158         else { pids[NW + i] = pid; }
159     }
160
161     for (int i = 0; i < (NW + NR); i++)
162     {
```

```
163             check_macros(pids[i]);
164         }
165 
166         if (shmdt(shmaddr) == -1) { perror("shmdt\n"); exit(1); }
167 
168         if (shmctl(shmid, IPC_RMID, (void*)shmaddr) < 0)
169         {
170             perror("rm shm error\n");
171             exit(1);
172         }
173 
174         if (semctl(semid, 0, IPC_RMID) < 0)
175         {
176             perror("rm sem error\n");
177             exit(1);
178         }
179 }
```

5.3 Читатели-писатели Windows

```
1  #include <windows.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <process.h>
5  #include <stdbool.h>
6
7  #define READERS 4
8  #define WRITERS 3
9
10 HANDLE canRead;
11 HANDLE canWrite;
12 HANDLE mutex;
13
14 LONG activeReaders = 0;
15 LONG waitingReaders = 0;
16 LONG activeWriter = FALSE;
17 LONG waitingWriters = 0;
18
19 int value = 0;
20
21 void startRead()
22 {
23     InterlockedIncrement(&waitingReaders);
24
25     if (waitingWriters || WaitForSingleObject(canWrite, 0) ==
       WAIT_OBJECT_0)
26         WaitForSingleObject(canRead, INFINITE);
27
28     WaitForSingleObject(mutex, INFINITE);
29     SetEvent(canRead);
30     InterlockedDecrement(&waitingReaders);
31     InterlockedIncrement(&activeReaders);
32     ReleaseMutex(mutex);
33 }
34
35 void stopRead()
36 {
37     InterlockedDecrement(&activeReaders);
38
```

```
39      if (activeReaders == 0)
40      SetEvent(canWrite);
41  }
42
43  void startWrite()
44  {
45      InterlockedIncrement(&waitingWriters);
46
47      if (activeWriter || WaitForSingleObject(canRead, 0) ==
            WAIT_OBJECT_0)
48          WaitForSingleObject(canWrite, INFINITE);
49
50      WaitForSingleObject(mutex, INFINITE);
51      InterlockedDecrement(&waitingWriters);
52      InterlockedExchange(&activeWriter, TRUE);
53      ReleaseMutex(mutex);
54  }
55
56  void stopWrite()
57  {
58      ResetEvent(canWrite);
59      InterlockedExchange(&activeWriter, FALSE);
60
61      if (waitingReaders > 0)
62          SetEvent(canRead);
63      else
64          SetEvent(canWrite);
65  }
66
67  void LogExit(const char *msg)
68  {
69      perror(msg);
70      ExitProcess(EXIT_SUCCESS);
71  }
72
73  DWORD Reader(PVOID param)
74  {
75      srand(GetCurrentThreadId());
76
77      for (int i = 0; i < 6; i++)
78      {
```

```
79          Sleep(rand() % 200 + 100);
80          startRead();
81          printf("Reader have got = %d\n", value);
82          stopRead();
83      }
84      return EXIT_SUCCESS;
85  }
86
87  DWORD Writer(PVOID param)
88  {
89      srand(GetCurrentThreadId());
90
91      for (int i = 0; i < 6; i++)
92      {
93          Sleep(rand() % 500);
94          startWrite();
95          value++;
96          printf("Writer have incremented = %d\n", value);
97          stopWrite();
98      }
99      return EXIT_SUCCESS;
100 }
101
102 int main(void)
103 {
104     setbuf(stdout, NULL);
105
106     DWORD thid[WRITERS + READERS];
107     HANDLE pthread[WRITERS + READERS];
108
109     if ((canRead = CreateEvent(NULL, FALSE, FALSE, NULL)) == NULL)
110         LogExit("Can't createEvent");
111
112     if ((canWrite = CreateEvent(NULL, TRUE, FALSE, NULL)) == NULL)
113         LogExit("Can't t createEvent");
114
115     if ((mutex = CreateMutex(NULL, 0, NULL)) == NULL)
116         LogExit("Can't  createMutex");
117
118     for (int i = 0; i < WRITERS; i++)
119     {
```

```
120         pthread[i] = CreateThread(NULL, 0, Writer, NULL, 0,
                &thid[i]);
121
122         if (pthread[i] == NULL)
123             LogExit("Can't  createThread");
124     }
125     for (int i = WRITERS; i < WRITERS + READERS; i++)
126     {
127         pthread[i] = CreateThread(NULL, 0, Reader, NULL, 0,
                &thid[i]);
128
129         if (pthread[i] == NULL)
130             LogExit("Can't  createThread");
131     }
132
133     for (int i = 0; i < WRITERS + READERS; i++)
134     {
135         DWORD dw = WaitForSingleObject(pthread[i], INFINITE);
136
137         switch (dw)
138         {
139             case WAIT_OBJECT_0:
140             printf("Thread %d finished\n", thid[i]);
141             break;
142             case WAIT_TIMEOUT:
143             printf("WaitThread timeout %d\n", dw);
144             break;
145             case WAIT_FAILED:
146             printf("WaitThread failed %d\n", dw);
147             break;
148             default:
149             printf("Unknown %d\n", dw);
150             break;
151         }
152     }
153     CloseHandle(canRead);
154     CloseHandle(canWrite);
155     CloseHandle(mutex);
156
157     return EXIT_SUCCESS;
158 }
```

5.4 Демон

```
1  #define LOCKFILE "/var/run/daemon.pid"
2  #define LOCKMODE (S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)
3
4  int lockfile(int fd)
5  {
6      struct flock fl;
7
8      fl.l_type = F_WRLCK;
9      fl.l_start = 0;
10     fl.l_whence = SEEK_SET;
11     fl.l_len = 0;
12
13     return fcntl(fd, F_SETLK, &fl);
14 }
15
16 void daemonize(const char *cmd)
17 {
18     int i, fd0, fd1, fd2;
19     pid_t pid;
20     struct rlimit rl;
21     struct sigaction sa;
22
23     umask(0);
24
25     if (getrlimit(RLIMIT_NOFILE, &rl) < 0)
26     {
27         fprintf(stderr, "%s: невозможно получить максимальный номе
               р дескриптора", cmd);
28         exit(1);
29     }
30
31     if ((pid = fork()) < 0)
32     {
33         fprintf(stderr, "%s: ошибка вызова функции fork", cmd);
34         exit(1);
35     }
36     else if (pid != 0)
37     {
38         exit(0);
```

```
39      }
40
41      setsid();
42      sa.sa_handler = SIG_IGN;
43      sigemptyset(&sa.sa_mask);
44      sa.sa_flags = 0;
45      if (sigaction(SIGHUP, &sa, NULL) < 0)
46      {
47          fprintf(stderr, "Невозможно игнорировать сигнал SIGHUP");
48          exit(1);
49      }
50
51      if (chdir("/") < 0)
52      {
53          fprintf(stderr, "Невозможно сделать текущим рабочим катало
                гом /");
54          exit(1);
55      }
56
57      if (rl.rlim_max == RLIM_INFINITY)
58      {
59          rl.rlim_max = 1024;
60      }
61      for (i = 0; i < rl.rlim_max; i++)
62      {
63          close(i);
64      }
65
66      fd0 = open("/dev/null", O_RDWR);
67      fd1 = dup(0);
68      fd2 = dup(0);
69
70      openlog(cmd, LOG_CONS, LOG_DAEMON);
71      if (fd0 != 0 || fd1 != 1 || fd2 != 2)
72      {
73          syslog(LOG_ERR, "Ошибочные файловые дескрипторы %d %d
                %d", fd0, fd1, fd2);
74          exit(1);
75      }
76  }
77
```

```
78 int already_running()
79 {
80     int fd;
81     char buf[16];
82
83     fd = open(LOCKFILE, O_RDWR|O_CREAT, LOCKMODE);
84     if (fd < 0)
85     {
86         syslog(LOG_ERR, "Невозможно открыть %s: %s", LOCKFILE,
              strerror(errno));
87         exit(1);
88     }
89     if (lockfile(fd) < 0)
90     {
91         if (errno == EACCES || errno == EAGAIN)
92         {
93             close(fd);
94             return 1;
95         }
96         syslog(LOG_ERR, "Невозможно установить блокировку на %s:
              %s", LOCKFILE, strerror(errno));
97         exit(1);
98     }
99     ftruncate(fd, 0);
100    sprintf(buf, "%ld", (long)getpid());
101    write(fd, buf, strlen(buf) + 1);
102
103    return 0;
104 }
105
106 sigset_t mask;
107
108 /* void reread() {} */
109
110 void *thr_fn(void *arg)
111 {
112     int err, signo;
113     for (;;)
114     {
115         err = sigwait(&mask, &signo);
116         if (err != 0)
```

```
117             {
118                 syslog(LOG_ERR, "Ошибка вызова функции sigwait");
119                 exit(1);
120             }
121             switch (signo)
122             {
123                 case SIGHUP:
124                     syslog(LOG_INFO, "Чтение конфигурационного файла
                        ");
125                     /* reread(); */
126                     break;
127                 case SIGTERM:
128                     syslog(LOG_INFO, "Получен сигнал SIGTERM; выход");
129                     exit(0);
130                 default:
131                     syslog(LOG_INFO, "Получен непредвиденный сигнал
                        %d\n", signo);
132             }
133         }
134         return 0;
135 }
136
137 int main(int argc, char *argv[])
138 {
139         int err;
140         pthread_t tid;
141         char *cmd;
142         struct sigaction sa;
143
144         if ((cmd = strrchr(argv[0], '/')) == NULL)
145             cmd = argv[0];
146         else
147             cmd++;
148
149         printf("Логин до daemonize: %s\n", getlogin());
150
151         daemonize(cmd);
152
153         if (already_running())
154         {
155             syslog(LOG_ERR, "Демон уже запущен");
```

```
156             exit(1);
157         }
158
159         sa.sa_handler = SIG_DFL;
160         sigemptyset(&sa.sa_mask);
161         sa.sa_flags = 0;
162         if (sigaction(SIGHUP, &sa, NULL) < 0)
163         {
164             fprintf(stderr, "Невозможно восстаносить действие SIG_DFL
                    для SIGHUP");
165             exit(1);
166         }
167         sigfillset(&mask);
168         if ((err = pthread_sigmask(SIG_BLOCK, &mask, NULL)) != 0)
169         {
170             fprintf(stderr, "Ошибка выполнения операции SIG_BLOCK");
171             exit(1);
172         }
173         err = pthread_create(&tid, NULL, thr_fn, 0);
174         if (err != 0)
175         {
176             fprintf(stderr, "Невозможно создать поток");
177             exit(1);
178         }
179
180         syslog(LOG_INFO, "Логин после daemonize: %s\n", getlogin());
181
182         for (;;)
183         {
184             time_t t = time(NULL);
185             struct tm tm = *localtime(&t);
186             syslog(LOG_INFO, "%s: Текущее время: %02d:%02d:%02d\n",
                        cmd, tm.tm_hour, tm.tm_min, tm.tm_sec);
187             sleep(1);
188         }
189
190         return 0;
191 }
```

5.5 Булочная, скелетон

```
1  struct REQUEST
2  {
3      int index;
4      int number;
5      int pid;
6  };
7
8  program BAKERY_PROG
9  {
10     version BAKERY_VER
11     {
12         struct REQUEST GET_NUMBER(struct REQUEST) = 1;
13         int BAKERY_SERVICE(struct REQUEST) = 2;
14     } = 1; /* Version number = 1 */
15 } = 0x20000001; /* RPC program number */
```

5.6 Булочная, клиент

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/wait.h>
#include "bakery.h"

void bakery_prog_1(char *host)
{
    CLIENT *clnt;
    struct REQUEST  *result_1;
    struct REQUEST  request;

    request.pid = getpid();

    int  *result_2;

#ifndef DEBUG
    clnt = clnt_create (host, BAKERY_PROG, BAKERY_VER, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif  /* DEBUG */

    srand(time(NULL));
    usleep(rand() % 1000 + 5000000);

    result_1 = get_number_1(&request, clnt);
    if (result_1 == (struct REQUEST *) NULL) {
        clnt_perror (clnt, "call failed");
    }
    request.index = result_1->index;
    request.number = result_1->number;
    printf("Client (pid: %d) got ticket %d\n", request.pid,
       request.number);

    usleep(rand() % 1000 + 5000000);
    result_2 = bakery_service_1(&request, clnt);
```

```
39      if (result_2 == (int *) NULL) {
40          clnt_perror (clnt, "call failed");
41      }
42      printf("Client (pid: %d) with ticket %d got served with:
            %c\n", request.pid, request.number, *result_2);
43
44  #ifndef DEBUG
45      clnt_destroy (clnt);
46  #endif    /* DEBUG */
47  }
48
49  int main(int argc, char *argv[])
50  {
51      char *host;
52      if (argc < 2) {
53          host = "localhost";
54      } else if (argc == 2) {
55          host = argv[1];
56      } else {
57          printf("usage: %s server_host\n", argv[0]);
58          exit(1);
59      }
60      printf("pid: %d\n", getpid());
61      bakery_prog_1(host);
62  exit (0);
63  }
```

5.7 Булочная, сервер

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <pthread.h>
4  #include "bakery.h"
5  #include <time.h>
6
7  #include <sys/syscall.h>
8
9  pid_t gettid()
10 {
11     return syscall(SYS_gettid);
12 }
13
14 static int cur = 0;
15 static char ch = 'a';
16 static int client_is_getting_ticket[128] = {0};
17 static int numbers[128] = {0};
18 static int pids[128] = {0};
19
20 int get_max_ticket_number()
21 {
22     int max_res = 0;
23     for (int i = 0; i < 128; i++)
24         if (numbers[i] > max_res)
25             max_res = numbers[i];
26     return max_res;
27 }
28
29 void *get_ticket(void *arg)
30 {
31     time_t mytime = time(NULL);
32     static struct REQUEST  result;
33     struct REQUEST *argp = arg;
34
35     argp->index = cur;
36     result.index = cur;
37     cur++;
38
39     result.pid = argp->pid;
```

```
40      pids[argp->index] = argp->pid;
41
42      client_is_getting_ticket[argp->index] = 1;
43      result.number = get_max_ticket_number() + 1;
44      numbers[argp->index] = result.number;
45      client_is_getting_ticket[argp->index] = 0;
46
47      struct tm *now = localtime(&mytime);
48      struct timeval tv;
49      gettimeofday(&tv, NULL);
50
51      printf("Thread %d gave ticket %d to client with pid %d at
            %02d:%02d:%02d:%03ld\n",
52              gettid(), result.number, result.pid,
53              now->tm_hour, now->tm_min, now->tm_sec, tv.tv_usec /
                    1000);
54
55      return &result;
56  }
57  struct REQUEST *get_number_1_svc(struct REQUEST *p_arg, struct
        svc_req *rqstp)
58  {
59      static struct REQUEST  result;
60      void *tmp;
61      int c;
62      pthread_t thread;
63      pthread_attr_t attr;
64
65      c = pthread_attr_init(&attr);
66      if (c != 0)
67      {
68          perror("pthread_attr_init");
69          exit(1);
70      }
71      c = pthread_create(&thread, &attr, get_ticket, p_arg);
72      if (c != 0)
73      {
74          perror("pthread_create");
75          exit(1);
76      }
77      c = pthread_attr_destroy(&attr);
```

```
78      if (c != 0)
79      {
80          perror("pthread_attr_destroy");
81          exit(1);
82      }
83      c = pthread_join(thread, &tmp);
84      if (c != 0)
85      {
86          perror("pthread_join");
87          exit(1);
88      }
89      result = *(struct REQUEST *)tmp;
90      return &result;
91  }
92  void *bakery_service(void *arg)
93  {
94      time_t mytime = time(NULL);
95
96      static int  result;
97      struct REQUEST *argp = arg;
98      for (int i = 0; i < 128; i++)
99      {
100         while (client_is_getting_ticket[i]) {}
101         while (numbers[i] != 0 && (numbers[i] <
                numbers[argp->index] ||
102                                    numbers[i] ==
                                       numbers[argp->index] &&
103                                    pids[i] < pids[argp->index]))
                                       {}
104     }
105     result = ch++;
106     numbers[argp->index] = 0;
107
108     struct tm *now = localtime(&mytime);
109     struct timeval tv;
110     gettimeofday(&tv, NULL);
111
112     printf("Thread␣%d␣served␣client␣with␣pid␣%d␣at␣
           %02d:%02d:%02d:%03ld␣with␣'%c'\n",
113             gettid(), pids[argp->index], now->tm_hour,
                now->tm_min, now->tm_sec, tv.tv_usec / 1000,
```

```
                result);
114
115     pthread_exit(&result);
116 }
117 int *bakery_service_1_svc(struct REQUEST *p_arg, struct svc_req
       *rqstp)
118 {
119     static int  result;
120     void *tmp;
121     int c;
122     pthread_t thread;
123     pthread_attr_t attr;
124
125     c = pthread_attr_init(&attr);
126     if (c != 0)
127     {
128         perror("pthread_attr_init");
129         exit(1);
130     }
131     c = pthread_create(&thread, &attr, bakery_service, p_arg);
132     if (c != 0)
133     {
134         perror("pthread_create");
135         exit(1);
136     }
137     c = pthread_attr_destroy(&attr);
138     if (c != 0)
139     {
140         perror("pthread_attr_destroy");
141         exit(1);
142     }
143     c = pthread_join(thread, &tmp);
144     if (c != 0)
145     {
146         perror("pthread_join");
147         exit(1);
148     }
149
150     result = *(int *)tmp;
151     return &result;
152 }
```