Взаимодействие процессов

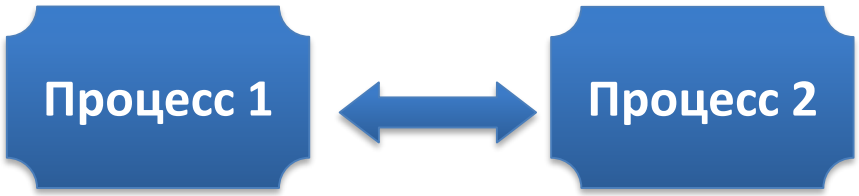
Ситуации, требующие взаимодействия процессов

Несколько процессов используют один и тот же ресурс.

Например, принтер или разделяемую память.

Требуется: определить порядок использования этого ресурса.

Процессы работают в разных адресных пространствах и не могут взаимодействовать друг с другом напрямую



Решение одной задачи выполняется несколькими процессами.

Требуется: реализовать механизм передачи данных между процессами.

Взаимодействие процессов. Передача данных

- каналы (программные, FIFO)
- очереди сообщений
- разделяемая память + семафоры
- механизм сокетов
- ActiveX (OLE)
- буфер обмена

Каналы

- Неименованные (программные) каналы

```
user@user-pc:~$ cat myfile | sort
```

```
#include <unistd.h>
int pipe (int *filedes);
```

filedes[0] – дескриптор для чтения из канала

filedes[1] – дескриптор для записи в канал

Неименованные (программные) каналы

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/wait.h>
extern int errno;

int main()
{
	int fd[2];
	int pid1, pid2, status;
	char buff[1];

	pipe(fd);

	pid1 = fork();
	pid2 = fork();

	if(pid1 == -1)
	{
		perror("Fork1");
		exit(errno);
	}
	if(pid2 == -1)
	{
		perror("Fork2");
		exit(errno);
	}
	if(pid1 == 0 && pid2 != 0)
	{
		printf("Child1\n");
		close(fd[0]);
		printf("Enter:\n");
		scanf("%c", buff);
		write(fd[1], buff, 1);
		close(fd[1]);
	}else if(pid2 == 0 && pid1 != 0)
	{
		printf("\nChild2\n");
		close(fd[1]);
		while(read(fd[0], buff, 1)>0)
		{
			printf("buff = %c\n",
				buff[0]);
		}
		close(fd[0]);
```

Неименованные (программные) каналы

```
        }else if(pid2 != 0 && pid1 != 0)
        {
                printf("Parent\n");
                wait(&status);
        }
        exit(0);
}//main
```

Каналы

- Именованные каналы. FIFO

```
#include <sys/type.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int mknod(char *pathname, mode_t mode, dev_t dev);
```

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *fifoname, mode_t mode);
```

Именованные каналы. FIFO

```
#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#define FIFO “fifo.1”
#define MAXBUFF 80

int main (void)
{
  int fd, n;
  char buff[MAXBUFF]; /*буфер для чтения данных */
  /*Создадим специальный файл FIFO */
  if (mknod(FIFO, S_IFIFO | 0666, 0) < 0){
    printf(“Невозможно создать FIFO\n”);
    exit(1);
  }
```

Именованные каналы. FIFO

```
/*Получим доступ к FIFO*/
  if ((fd = open(FIFO, O_RDONLY)) < 0){
    printf("Невозможно открыть FIFO\n");
    exit(1);
  }
  /*Прочитаем сообщение и выведем его на экран */
  while ((n = read(fd, buff, MAXBUFF)) > 0)
    if (write(1, buff, n) != n){
      printf("Ошибка вывода\n");
      exit(1);
    }
  /* Закроем FIFO, и удалим файл */
  close(fd);
  if (unlink(FIFO) < 0){
    printf("Невозможно удалить FIFO\n"); exit(1);
  }
  exit(0);
}
```

Именованные каналы. FIFO

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
/*Соглашение об имени FIFO*/
#define FIFO “fifo.1”

int main (void){
  int fd, n;
  /*Получим доступ к FIFO*/
  if ((fd = open(FIFO, O_WRONLY)) < 0){
    printf(“Невозможно открыть FIFO\n”);
    exit(1);
  }
  /*Передадим сообщение серверу FIFO*/
  if (write(fd, “Hello, World!\n\0”, 15) != 15){
    printf(“Ошибка записи\n”); exit(1);
  }
  close(fd);
  exit (0);
}
```

Передача сообщений

Этот механизм использует операции: ***send*** и ***receive***.

Процесс-отправитель посылает сообщение заданному адресату, а процесс-получатель получает сообщение от указанного источника. Если сообщения нет, процесс-получатель блокируется до поступления сообщения либо немедленно возвращает код ошибки.

Чтобы избежать потери сообщений, при получении сообщения получатель посылает обратно ***код подтверждения***.

Передача сообщений часто используется в системах с параллельным программированием. Пример: MPI (Message-Passing Interface – интерфейс передачи сообщений).

Состояние состязания

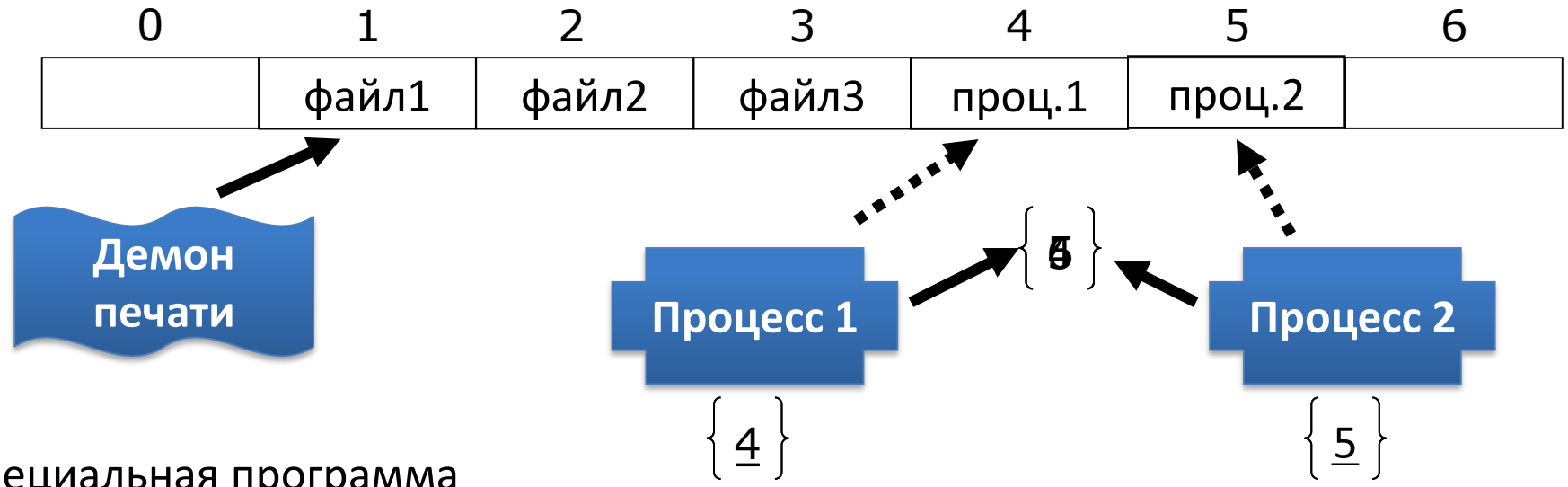
Ситуация, в которой два (и более) процесса считывают или записывают данные одновременно, и конечный результат зависит от того, какой из них был первым, называется ***состоянием состязания.***

Как избежать состязания?

Запрет одновременной записи и чтения разделённых данных более чем одним процессом => ***взаимное исключение!***

Состояние состязания

Имеется разделяемый ресурс – очередь печати файлов.



Специальная программа выбирает файлы из очереди, печатает их на принтере и удаляет из очереди

Для указания очередной свободной ячейки в очереди используется разделяемая переменная. Процессы сначала определяют значение этой переменной, записывают его в свои локальные переменные, затем записывают имя файла в соответствующую ячейку.

Критическая область

Критическая область (критическая секция) – часть программы, в которой есть обращение к совместно используемым данным

Условия правильной совместной работы процессов:

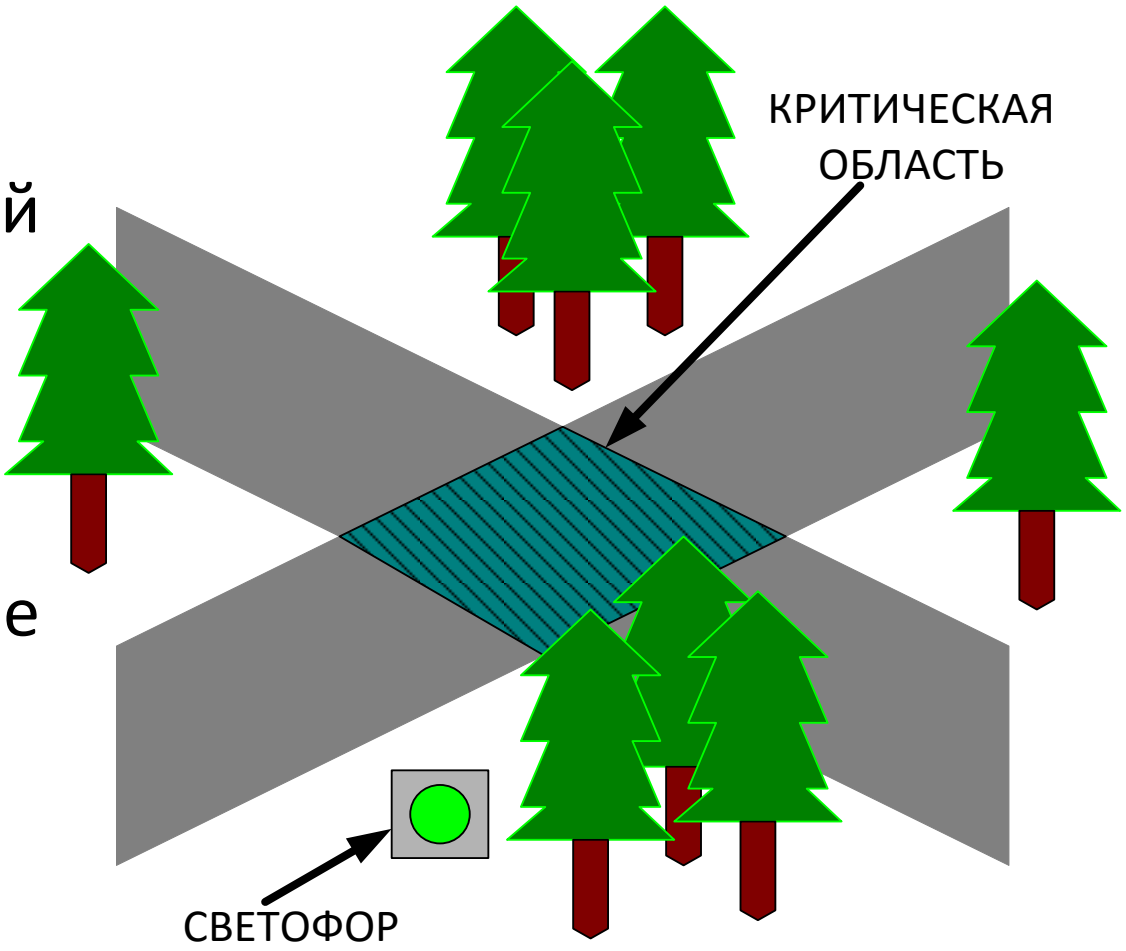
- два процесса не должны одновременно находиться в критических областях
- в программе не должно быть предположений о количестве и скорости процессов
- процесс, находящийся вне критической области, не может блокировать другие процессы
- недопустима ситуация, в которой процесс вечно ждёт попадания в критическую секцию

Взаимное исключение с использованием критических областей

Условие разных скоростей процессов

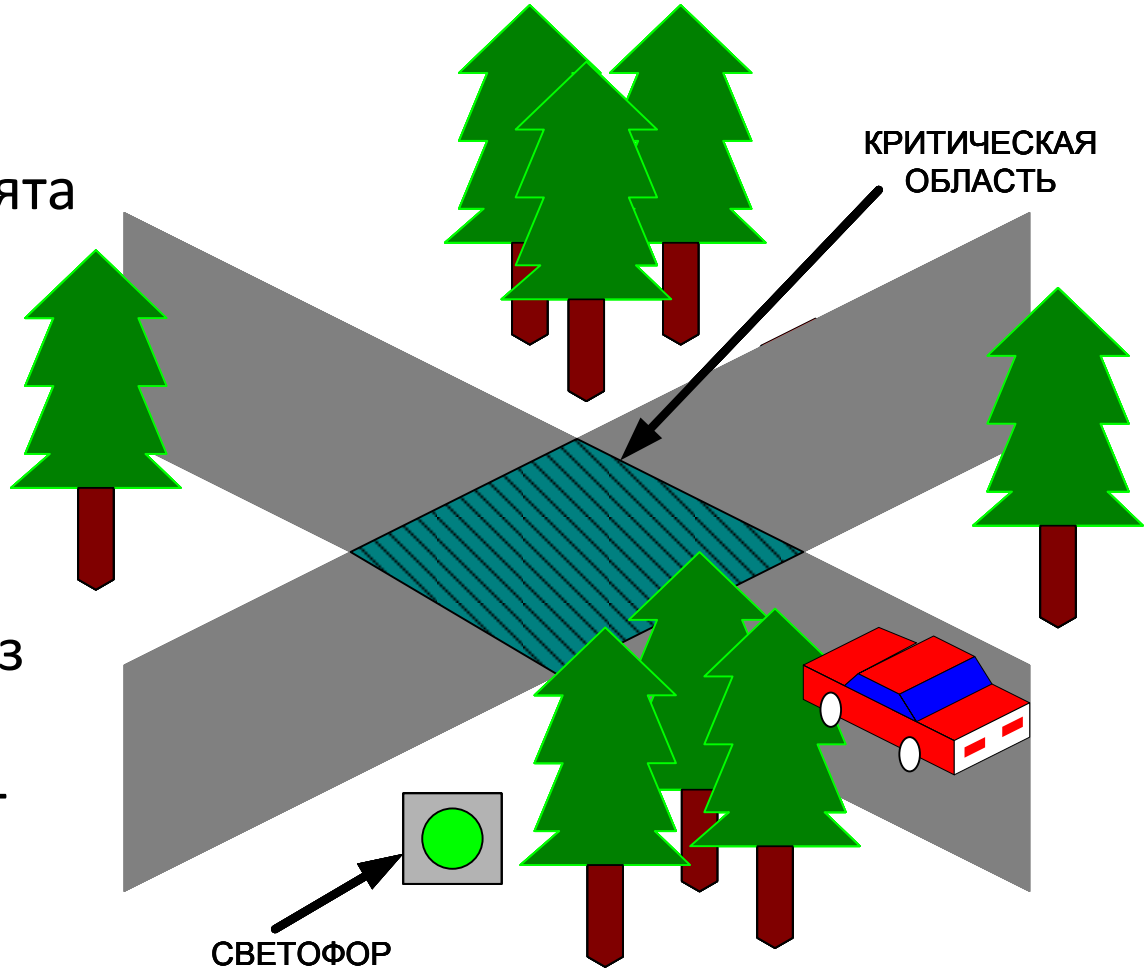
Имеется критический участок.

Требуется организовать его взаимоисключающее использование.



Принцип работы взаимоисключения

Процесс, желающий попасть в критическую область проверяет занята ли она. Если нет, то он входит в неё и сигнализирует об этом другим процессам, которым приходится ждать. После выхода из критической области процесс сигнализирует об этом. Ситуация повторяется с другим процессом.



Способы реализации взаимного исключения

Для того, чтобы обеспечить нахождения только одного процесса в критической области каждый процесс обязан:

При входе:

- ✓ проверить, есть ли кто в критической области;
- ✓ объявить о том, что он находится в критической области (запретив при этом доступ в критическую область другим процессам).

При выходе сообщить, что в критической области больше никого нет (разрешив тем самым доступ к критической области другим процессам).

Если при проверке доступности критической секции процесс обнаружил, что она недоступна, то он может:

- ✓ продолжать проверять доступность критической секции;
- ✓ заблокироваться и ждать сообщения от операционной системы, что критическая область освободилась.

Первый способ называется ***активным ожиданием (спин-блокировкой)*** и сопровождается нерациональным использованием процессора.

Алгоритм Деккера

Деккер (T. Dekker) разработал программное решение проблемы взаимного исключения, не требующее строго чередования:

```
1  # define FALSE 0
2  # define TRUE  1
3  # define N 2                    /* Количество процессов */
4  int turn = 0;                          /* Чья сейчас очередь?*/
5  int interested[N] = {0,0};
6  void enter_region(int process)
7  {
8    int other;                           /* Номер 2-го процесса */
9    other = 1 - process;
10   while(TRUE)
11   {
12     interested[process] = TRUE;     /* Индикатор интереса */
13     do {
14       if (!interested[other]) return;
15     } while (turn != other);
16     interested[process] = FALSE;
17     while (turn == other);
18   }
19 }
20 void leave_region (int process)
21 {
22   turn = 1 - process;
23   interested[process] = FALSE;       /* Индикатор выхода из
24                                    критической области */
25 }
```

```
#!/usr/bin/bash

NAME = file25 ; SHOWALL=0
printf “Введите суффикс” ; read $SUFFIX
case $SUFFIX in
 *) NAME=$SUFFIX.$NAME
    ;;
 ?) NAME=$NAME”.”SUFFIX
    ;;
 c) NAME=$NAMEA.c
    ;;
esac
if [ -f $NAME ] ; then
  . $NAME
fi
```

```
user@user-pc:~/Test$ ./kr1.sh
bash: ./kr1.sh: /usr/bin/bash: плохой интерпретатор: Нет такого
файла или каталога

user@user-pc:~/Test$ whereis bash
bash: /bin/bash /etc/bash.bashrc /usr/share/man/man1/bash.1.gz
```

```
#!/bin/bash

NAME = file25 ; SHOWALL=0
printf “Введите суффикс” ; read $SUFFIX
case $SUFFIX in
 *) NAME=$SUFFIX.$NAME
    ;;
 ?) NAME=$NAME”.”SUFFIX
    ;;
 c) NAME=$NAMEA.c
    ;;
esac
if [ -f $NAME ] ; then
 . $NAME
fi
```

```
user@user-pc:~/Test$ ./kr1.sh
./kr1.sh: строка 2: NAME: команда не найдена
```

```
#!/bin/bash

NAME=file25 ; SHOWALL=0
printf “Введите суффикс” ; read $SUFFIX
case $SUFFIX in
 *) NAME=$SUFFIX.$NAME
    ;;
 ?) NAME=$NAME”.”SUFFIX
    ;;
 c) NAME=$NAMEA.c
    ;;
esac
if [ -f $NAME ] ; then
 . $NAME
fi
```

```
#!/bin/bash

NAME=file25 ; SHOWALL=0
printf “Введите суффикс ” ; read SUFFIX
case $SUFFIX in
 *) NAME=$SUFFIX.$NAME
    ;;
 ?) NAME=$NAME“.”SUFFIX
    ;;
 c) NAME=$NAMEA.c
    ;;
esac
echo $NAME
if [ -f $NAME ] ; then
  . $NAME
fi
```

```
user@user-pc:~/Test$ ./kr1.sh
Введите суффикс c
c.file25
```

```
#!/bin/bash

NAME=file25 ; SHOWALL=0
printf “Введите суффикс ” ; read SUFFIX
case $SUFFIX in
 c) NAME=$NAMEA.c
    ;;
 *) NAME=$SUFFIX.$NAME
    ;;
 ?) NAME=$NAME“.”SUFFIX
    ;;
esac
echo $NAME
if [ -f $NAME ] ; then
  . $NAME
fi
```

```
user@user-pc:~/Test$ ./kr1.sh
Введите суффикс c
.c
```

```
#!/bin/bash

NAME=file25 ; SHOWALL=0
printf “Введите суффикс ” ; read SUFFIX
case $SUFFIX in
 c) NAME=$NAME“A.c” NAME=$NAME.c NAME=$NAME“.c”
    ;;
 *) NAME=$SUFFIX.$NAME
    ;;
 ?) NAME=$NAME“.”SUFFIX
    ;;
esac
echo $NAME
if [ -f $NAME ] ; then
 . $NAME
fi
```

```
user@user-pc:~/Test$ ./kr1.sh
Введите суффикс c
file25A.c
```

```
#!/bin/bash

NAME=file25 ; SHOWALL=0
printf “Введите суффикс ” ; read SUFFIX
case $SUFFIX in
 c) NAME=$NAME“A.c”
    ;;
 *) NAME=$SUFFIX.$NAME
    ;;
 ?) NAME=$NAME“.”SUFFIX
    ;;
esac
echo $NAME
if [ -f $NAME ] ; then
 . $NAME
fi
```

```
user@user-pc:~/Test$ ./kr1.sh
Введите суффикс a
a.file25A

Введите суффикс abcd
abcd.file25A
```

```
#!/bin/bash

NAME=file25 ; SHOWALL=0
printf “Введите суффикс ” ; read SUFFIX
case $SUFFIX in
 c) NAME=$NAME“A.c”
    ;;
 ?) NAME=$NAME“.”SUFFIX
    ;;
 *) NAME=$SUFFIX.$NAME
    ;;
esac
echo $NAME
if [ -f $NAME ] ; then
 . $NAME
fi
```

```
user@user-pc:~/Test$ ./kr1.sh
Введите суффикс a
file25.SUFFIX

Введите суффикс abcd
abcd.file25
```

```
#!/bin/bash

NAME=file25 ; SHOWALL=0
printf “Введите суффикс ” ; read SUFFIX
case $SUFFIX in
 c) NAME=$NAME“A.c”
    ;;
 ?) NAME=$NAME“.”$SUFFIX
    ;;
 *) NAME=$SUFFIX.$NAME
    ;;
esac
echo $NAME
if [ -f $NAME ] ; then
  . $NAME
fi
```

```
user@user-pc:~/Test$ ./kr1.sh
Введите суффикс c
file25A.c
Hello! I am file25A.c
```

Содержимое файла file25A.c

```
echo “Hello I am file25A.c”
```

Способы реализации взаимного исключения

Для того, чтобы обеспечить нахождения только одного процесса в критической области каждый процесс обязан:

При входе:

- ✓ проверить, есть ли кто в критической области;
- ✓ объявить о том, что он находится в критической области (запретив при этом доступ в критическую область другим процессам).

При выходе сообщить, что в критической области больше никого нет (разрешив тем самым доступ к критической области другим процессам).

Если при проверке доступности критической секции процесс обнаружил, что она недоступна, то он может:

- ✓ продолжать проверять доступность критической секции;
- ✓ заблокироваться и ждать сообщения от операционной системы, что критическая область освободилась.

Первый способ называется ***активным ожиданием (спин-блокировкой)*** и сопровождается нерациональным использованием процессора.

Алгоритм Деккера

Деккер (T. Dekker) разработал программное решение проблемы взаимного исключения, не требующее строго чередования:

```
1  # define FALSE 0
2  # define TRUE  1
3  # define N 2                    /* Количество процессов */
4  int turn = 0;                          /* Чья сейчас очередь?*/
5  int interested[N] = {0,0};
6  void enter_region(int process)
7  {
8    int other;                           /* Номер 2-го процесса */
9    other = 1 - process;
10   while(TRUE)
11   {
12     interested[process] = TRUE;     /* Индикатор интереса */
13     do {
14       if (!interested[other]) return;
15     } while (turn != other);
16     interested[process] = FALSE;
17     while (turn == other);
18   }
19 }
20 void leave_region (int process)
21 {
22   turn = 1 - process;
23   interested[process] = FALSE;       /* Индикатор выхода из
24                                    критической области */
25 }
```

Алгоритм Петерсона

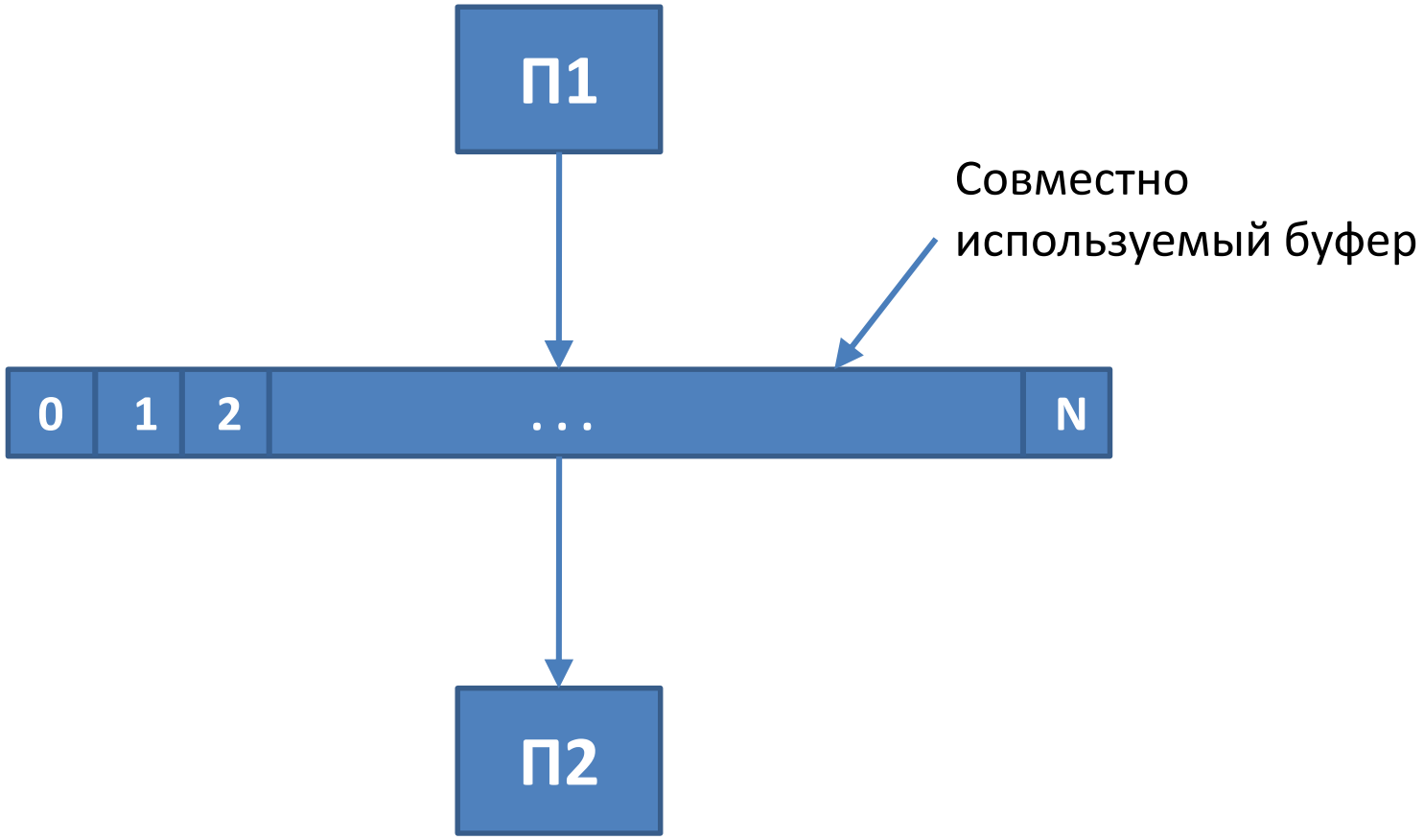
В 1981 г. Петерсон (G.L. Peterson) разработал существенно более простой алгоритм взаимного исключения:

```
# define FALSE 0
# define TRUE  1
# define N 2                          /* Количество процессов */
int turn;                                  /* Чья сейчас очередь?*/
int interested [N] = {0,0};

void enter_region (int process)
{
   int other;                         /* Номер 2-го процесса */
   other = 1 - process;
   interested [process] = TRUE;    /* Индикатор интереса */
   turn = process;
   while (turn == process && interested [other] == TRUE);
}

void leave_region (int process)
{
   interested [process] = FALSE;  /* Индикатор выхода из
                                     критической области */
}
```

Проблема производителя и потребителя



Проблема ограниченного буфера

```
#define N 100   /* Максимальное количество элементов в буфере */
int count = 0;  /* Текущее количество элементов в буфере */

void producer()
{
   int item;
   while(TRUE)
   {
     item = produce_item();
     if(count == N) sleep();/*Если буфер полон, уйти в состояние
                              ожидания*/
     insert_item(item);
     count = count + 1;
     if(count == 1) wakeup(consumer);
   }
}
void consumer()
{
   int item;
   while(TRUE)
   {
     if(count == 0) sleep(); /*Если буфер пуст, уйти в состояние
                               ожидания*/
     item = remove_item();
     count = count - 1;
     if(count == (N - 1)) wakeup(producer);
     consume_item(item);
   }
```

Семафоры

Концепцию семафоров предложил Дейкстра (E.W. Dijkstra) в 1965 году.

Семафоры – переменные, использующиеся для подсчёта сигналов запуска, сохранённых на будущее.

Значение семафоров может быть 0 (в случае отсутствия сохранённых сигналов активизации) или некоторым положительным числом, соответствующим количеству отложенных активизирующих сигналов.

С семафорами связаны две операции:

- **down** (P):
  - сравнение значения семафора с 0
  - если значение больше 0, оно уменьшается на 1 и управление возвращается
  - если значение равно 0, управление процессу не возвращается, процесс переводится в состояние ожидания
- **up** (V):
  - увеличение значения семафора
  - активизация процесса, связанного с этим семафором

Решение проблемы производителя и потребителя с помощью семафоров

```
#define N 100                                   /* Количество сегментов в буфере*/

typedef int semaphore;
semaphore mutex = 1;                            /* Контроль доступа в критическую область */

semaphore empty = N;                            /* Число пустых сегментов в буфере */

semaphore full = 0;                             /* Число полных сегментов в буфере */

void producer(void)
{
   int item;
   while (TRUE)
   {
      item = produce_item();
      down(&empty);                             /* Уменьшить счётчик пустых сегментов */

      down(&mutex);                             /* Вход в критическую область */
      insert_item(item);

      up(&mutex);                               /* Выход из критической области */
      up(&full);                                /* Увеличить счётчик полных сегментов *
    }
}
```

Мьютекс **(mutex, сокращение от mutual exclusion – взаимное исключение) – упрощённая версия семафора.**

Мьютекс – переменная, которая может находиться в одном из двух состояний: блокированном или небло-кированном

Решение проблемы производителя и потребителя с помощью семафоров (продолжение)

```
void consumer (void)
{
   int item;
   while (TRUE)
   {
     down(&full);                  /* Уменьшить счётчик полных
                                       сегментов */
     down(&mutex);                 /* Вход в критическую область */
     item = remove_item();
     up(&mutex);                   /* Выход из критической области */
     up(&empty);                   /* Увеличить счётчик пустых
                                       сегментов */
     consume_item(item);
   }
}
```