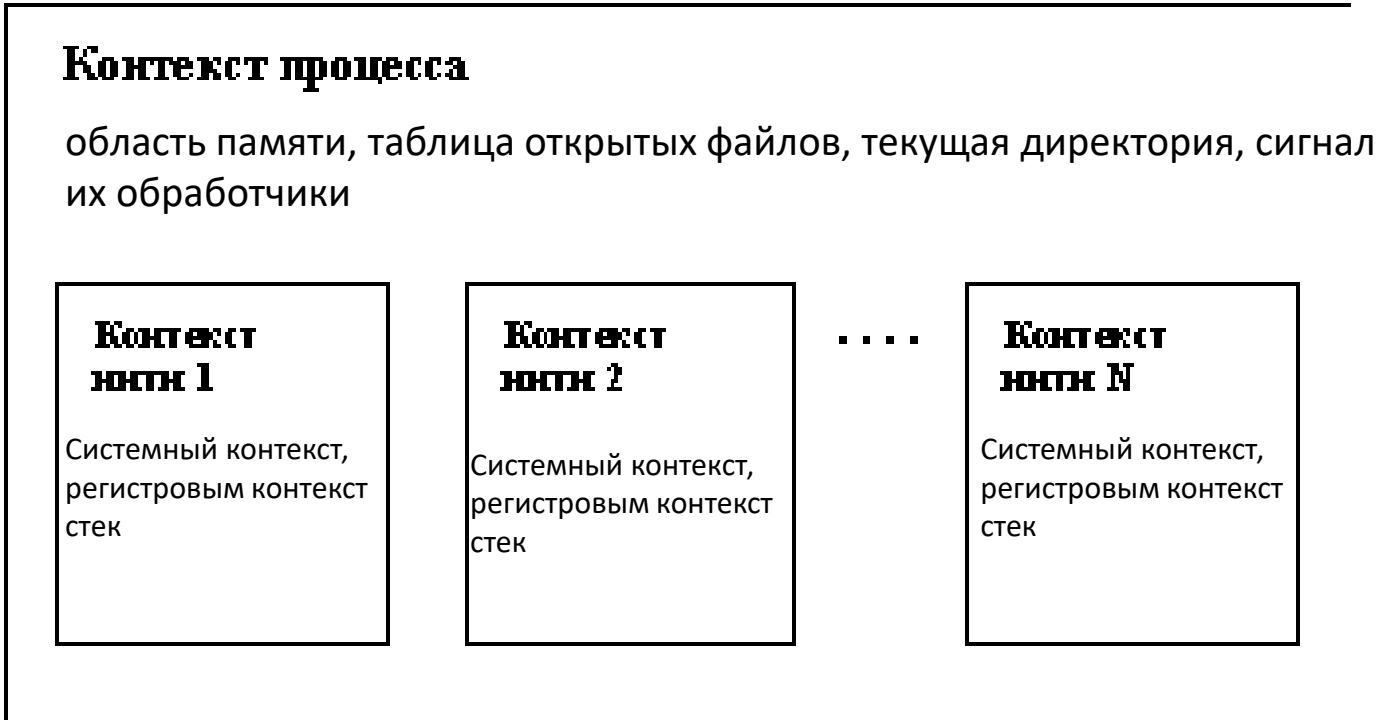
Соотношение контекста процесса и контекстов нитей

Нить (Поток)— это процесс, который использует ресурсы совместно с другими процессами. Каждый поток имеет дескриптор и представляется для ядра обычным процессом.
Максимальное число потоков:
max_threads = mempages / (8 * THREAD_SIZE / PAGE_SIZE), но не менее 20



Преимущества потоков перед процессами

- меньше времени для создания нового потока, поскольку создаваемый поток использует адресное пространство текущего процесса;
- меньше времени для завершения потока;
- меньше времени для переключения между двумя потоками в пределах процесса;
- меньше коммуникационных расходов, поскольку потоки разделяют все ресурсы, и в частности адресное пространство. Данные, продуцируемые одним из потоков, немедленно становятся доступными всем другим потокам.

Закон Амдаля для распараллеливания нитий

$$\text{Ускорение} = \frac{1}{F + (1 - F)/N}$$

N – число процессоров

F - указывает, какая часть системы не может быть распараллелена

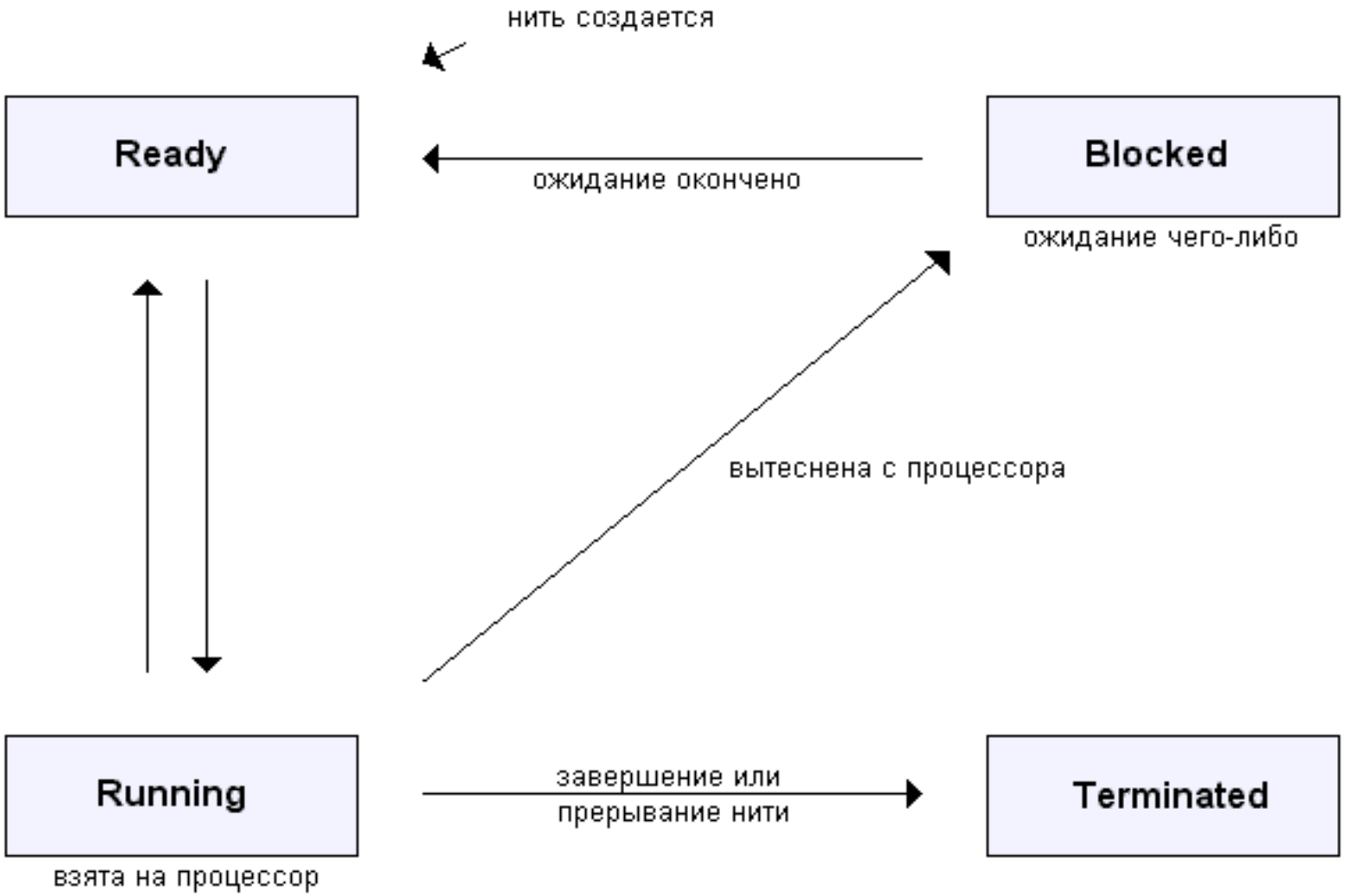
При N=2 и F=75% кода запускается параллельно, а 25% — последовательно

Ускорение=1.6

При N=4, Ускорение=2.28

При $N \to \infty$, Ускорение=4

Диаграмма состояний нити



Реализация потоков (нитей) при помощи функции **clone**

В отличии от fork() позволяет настраивать запускаемый процесс

int clone (int (*fn) (void *), void *child_stack, int flags, void *arg);

В случае успеха, возвращается PID нити,в случае ошибки -1

fn – указатель на функцию потока (имя функции);

child_stack – указатель на стек потока;

flags – флаги;

arg – аргумент, передаваемый функции.

Основные флаги системного вызова clone ()

Флаг	Описание
CLONE_FILES	Родительский и порожденный процессы совместно используют одну и ту же таблицу файловых дескрипторов (копия)
CLONE_FS	Родительский и порожденный процессы совместно используют информацию о файловой системе (копия)
CLONE_NEWNS	Создать новое пространство имен для порожденной задачи, только привилегированный процесс
CLONE_PARENT	Родительский процесс вызывающего процесса становится родительским и для порожденного (родителем будет вызывающий процесс)
CLONE_SIGHAND	У порожденного и родительского процессов будут общие обработчики сигналов (копия)
CLONE_THREAD	Родительский и порожденный процессы будут принадлежать одной группе потоков (в новой группе)
CLONE_VFORK	Использовать v*fork ()*: родительский процесс будет находиться в приостановленном состоянии, пока порожденный процесс не возобновит его работу (параллельная работа)
CLONE_VM	У порожденного и родительского процессов будет общее адресное пространство (разное)
SIGCHLD	Номер сигнала, который посылается родителю, когда потомок умирает

Шаблон создания потока с помощью clone

```
//стек размером 64kB
#define STACK 1024*64
#include </usr/include/linux/sched.h>
// Дочерний поток выполнит эту функцию
void Function( void* argument ) {
    ……….
  }
int main() {
    void* stack= malloc(STACK);
    pid_t pid;
    int  arg;
…….
 // Вызов clone() для создания дочернего потока
    pid = clone(&Function, (char*) stack + STACK,
    SIGCHLD | CLONE_FS | CLONE_FILES | CLONE_SIGHAND | CLONE_VM,  (void*)arg);
    ……..
 // Освободить память, используемую для стека
    free(stack);
    return 0;
}
```

Низкоуровневая поддержка потоков

#include <pthread.h>
int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);

В случае успешного выполнения функция возвращает 0. Если произошли ошибки, то могут быть возвращены следующие значения:

- **EAGAIN** – у системы нет ресурсов для создания нового потока, или система не может больше создавать потоков, так как количество потоков превысило значение PTHREAD_THREADS_MAX
- **EINVAL** – неправильные атрибуты потока, переданные аргументом attr
- **EPERM** – вызывающий поток не имеет должных прав для того, чтобы задать нужные параметры или политики планировщика.

Выходной параметр thread – идентификатор созданного потока

Новый поток будет выполнять функцию **start_routine** с прототипом

void * имя функции(void * arg);

- attr – атрибуты потока (область видимости; размер стека; адрес стека; приоритет; состояние; стратегия планирования и параметры). Если NULL, то атрибуы по умолчанию.

arg – значение, передаваемое в функцию. Так как функция может получать только указатель типа void, то все аргументы следует упаковать в структуру.

Шаблон создания потока с помощью pthread_create

```
#include <pthread.h>
 //потоковая функция
void* threadFunc(void* thread_data){
//завершаем поток
pthread_exit(0);
}
 int main(){
 //какие то данные для потока (для примера)
void* thread_data = NULL;

//создаем идентификатор потока
pthread_t thread;

//создаем поток по идентификатору thread и функции потока threadFunc
//и передаем потоку указатель на данные thread_data
pthread_create(&thread, NULL, threadFunc, thread_data);
//ждем завершения потока
pthread_join(thread, NULL);
 return 0;
}
```

Ожидания завершения потока

int pthread_join (pthread_t *thread,* void ***value* _ptr);

Приостанавливает выполнение текущего потока до тех пор, пока не завершится заданный поток *thread (идентификатор потока)*

При успешном завершении функция pthread_join() возвращает нулевое значение; в противном случае — код ошибки

Если параметр *value* _ptr не **NULL**, то по заданному указателю будет доступно значение кода завершения потока

Планировщик процессов

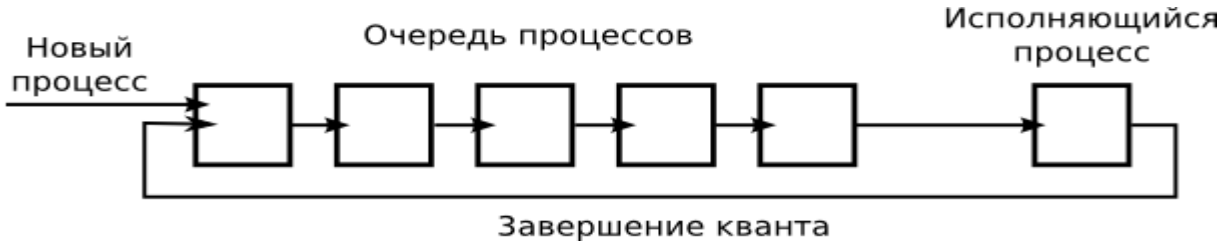
Планировщик *(scheduler) — это компонент ядра, который выбирает из всех про*цессов системы тот, который должен выполняться следующим на процессоре.

Планировщик вызывается:

1. При завершении или остановке процесса.
2. Когда истек квант времени, выделенный процессу.
3. При переходе процесса из состояния ожидания в состояния готовности к выполнению.

Стратегии планирования процессов

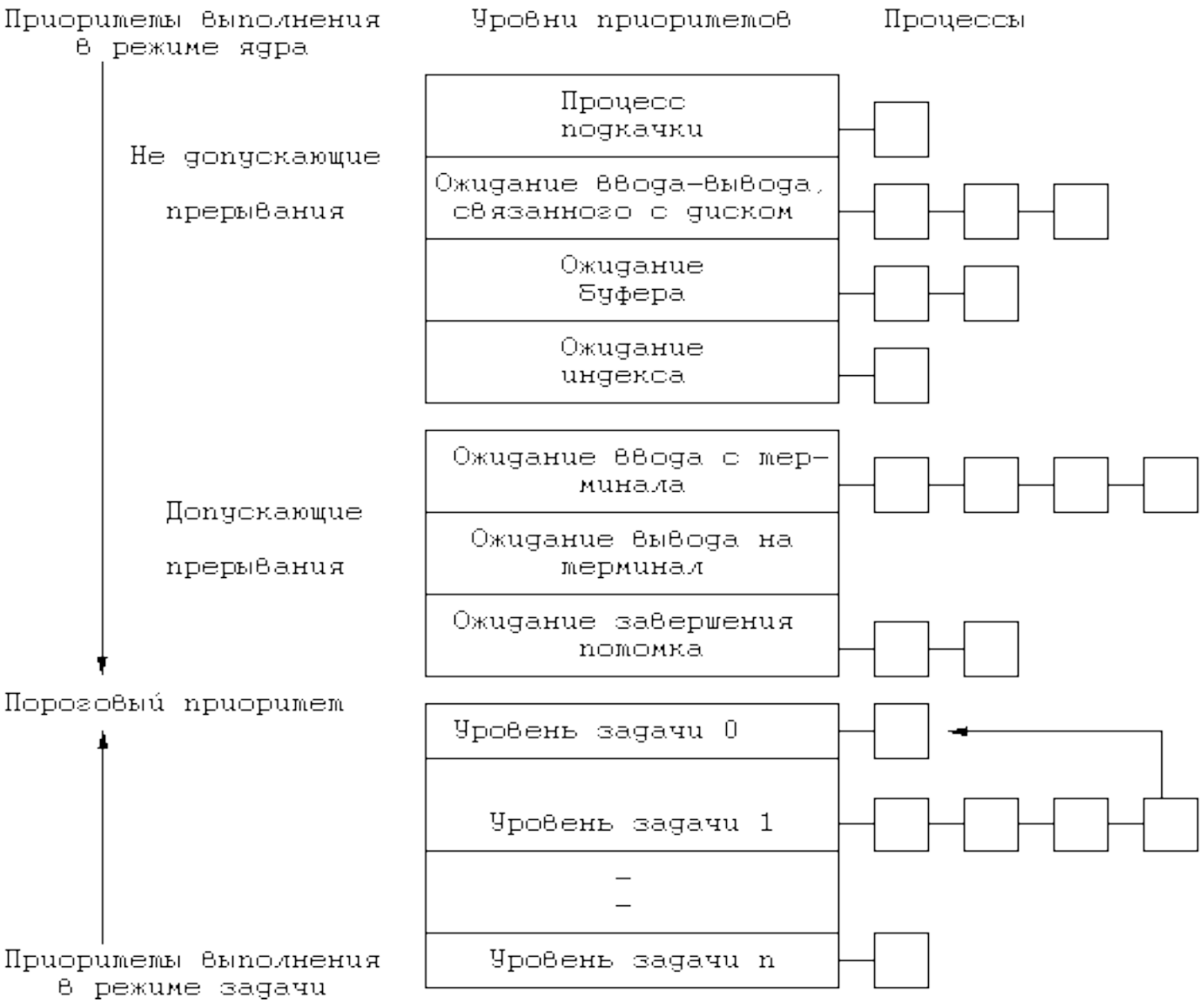
- SCHED_FIFO : политика планирования реального времени первый вошёл, первый вышел. Процесс выполняется до завершения, если он не заблокирован запросом ввода/вывода, вытеснен высокоприоритетным процессом, или он добровольно отказывается от процессора.
- SCHED_RR: циклическая (Round-Robin) политика планирования реального времени. Процессу разрешено работать максимум время кванта. Если процесс исчерпывает свой квант времени, он помещается в конец списка с его приоритетом.



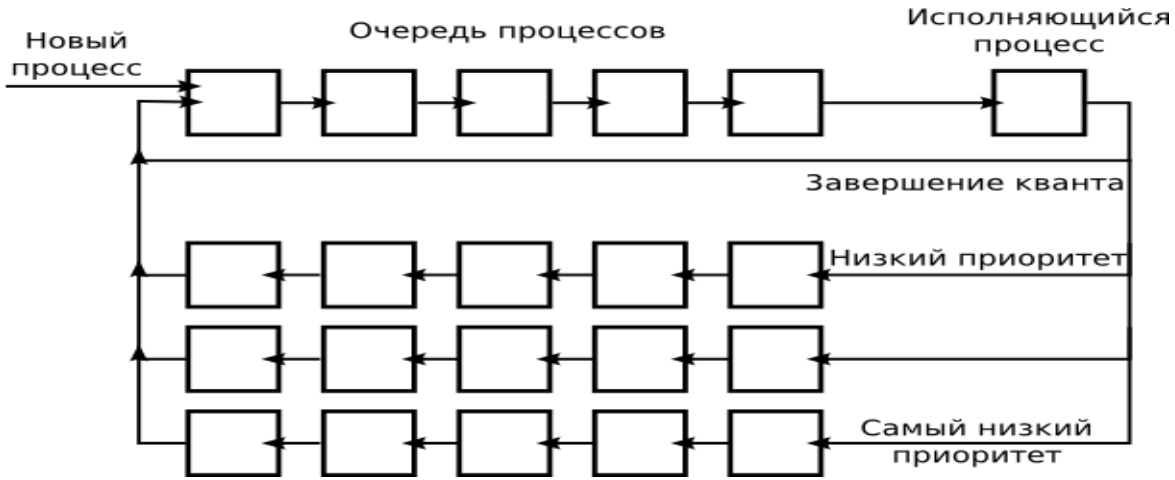
- SCHED_OTHER: политика планирования с разделением времени (политика по умолчанию), время выполнения определяется динамическим приоритетом

Дочерние процессы наследуют политику предка

Распределение задач по приоритетам



Общий алгоритм диспетчеризации




```
алгоритм schedule_process
{
выполнять пока (для запуска не будет выбран один из процессов)
    {
 для (каждого процесса в очереди готовых к выполнению)
        выбрать процесс с наивысшим приоритетом из загруженных в память;
если (ни один из процессов не может быть избран для выполнения)
приостановить машину; /* машина выходит из состояния простоя по прерыванию */
     }
удалить выбранный процесс из очереди готовых к выполнению;
переключиться на контекст выбранного процесса, возобновить его выполнение;
}
```

Приоритеты процесса

- *Статический приоритет*. Этот приоритет не изменяется с течением времени приоритет задается при создании процесса (по умолчанию 0, может быть изменен командой nice или renice , может быть изменен только явно пользователем. Он указывает максимальный размер временного кванта, который может быть выделен процессу, прежде чем другим процессам будет разрешено конкурировать за доступ к процессору.
- *Динамический приоритет*. Этот приоритет снижается с течением времени, пока процесс используется время процессора, меняется путем вычисления надбавки или штрафа в диапазоне от -5 до 5; когда его значение падает ниже 0, процесс помечается для повторного планирования. Это значение указывает остаток времени данного временного кванта.
- *Приоритет реального времени*. Этот приоритет показывает, какие другие процессы данный процесс побеждает в соревновании за время центрального процессора: более высокие значения всегда побеждают более низкие.

Диапазон приоритетов в пространстве пользователя

Стратегия планирования	Диапазон внутренних динамических приоритетов (s)	Диапазон статических приоритетов	Квант времени мс
SCHED_OTHER	99 – 139, по умолчанию 100	-20 до +19	20(140-s) s<120 5(140-s) s≥120 800…100…5
SCHED_FIFO	0 - 98	99 - 1	бесконечный
SCHED_RR	0 - 98	99 - 1	200…100…10

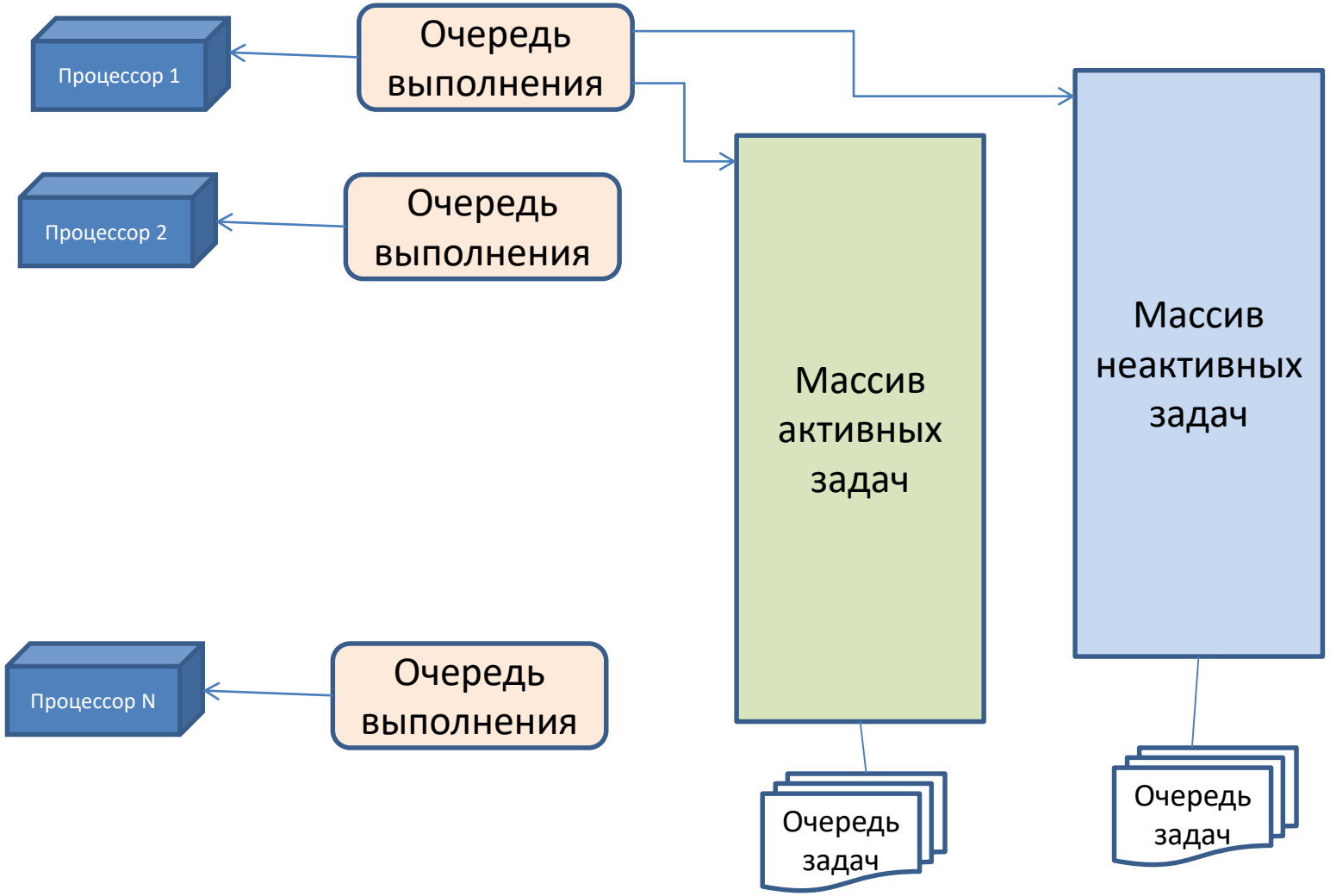
Настройка политики планирования процесса

chrt -m – вывод допустимых приоритетов;
chrt -p pid – вывод политики и приоритета;
chrt-p prio pid– установить приоритет;
chrt-f-p [1..99] pid – установка политики SCHED_FIFO;
chrt-o-p 0 pid – установка политики SCHED_OTHER;
chrt-r-p [1..99] pid – установка политики SCHED_RR

Функции планирования

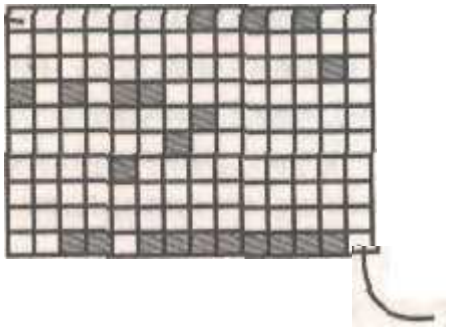
Метод	Описание
int sched_getscheduler (pid)	Получение класса планирования процесса. SCHED_FIFO =0, SCHED_RR=1, SCHED_OTHER =2
sched_setscheduler	Установка класса планирования процесса. (суперпользователь)
int sched_getparam (pid, struct sched_param *p) или getpriority (PRIO_PROCESS,pid)	Получение в поле **int** *sched_priority* структуры статического приоритета процесса.
sched_setparam или setpriority	Установка статического приоритета процесса. (суперпользователь)
int sched_get_priority_max (int *policy*)	Получение максимального разрешённого значения статического приоритета для класса планирования.
int sched_get_priority_min (int *policy*)	Получение минимального разрешённого значения статического приоритета для класса планирования.
sched_rr_get_interval	Получение текущего временного интервала для процесса SCHED_RR.
sched_yield	Передача выполнения другому процессу.

Архитектура планировщика

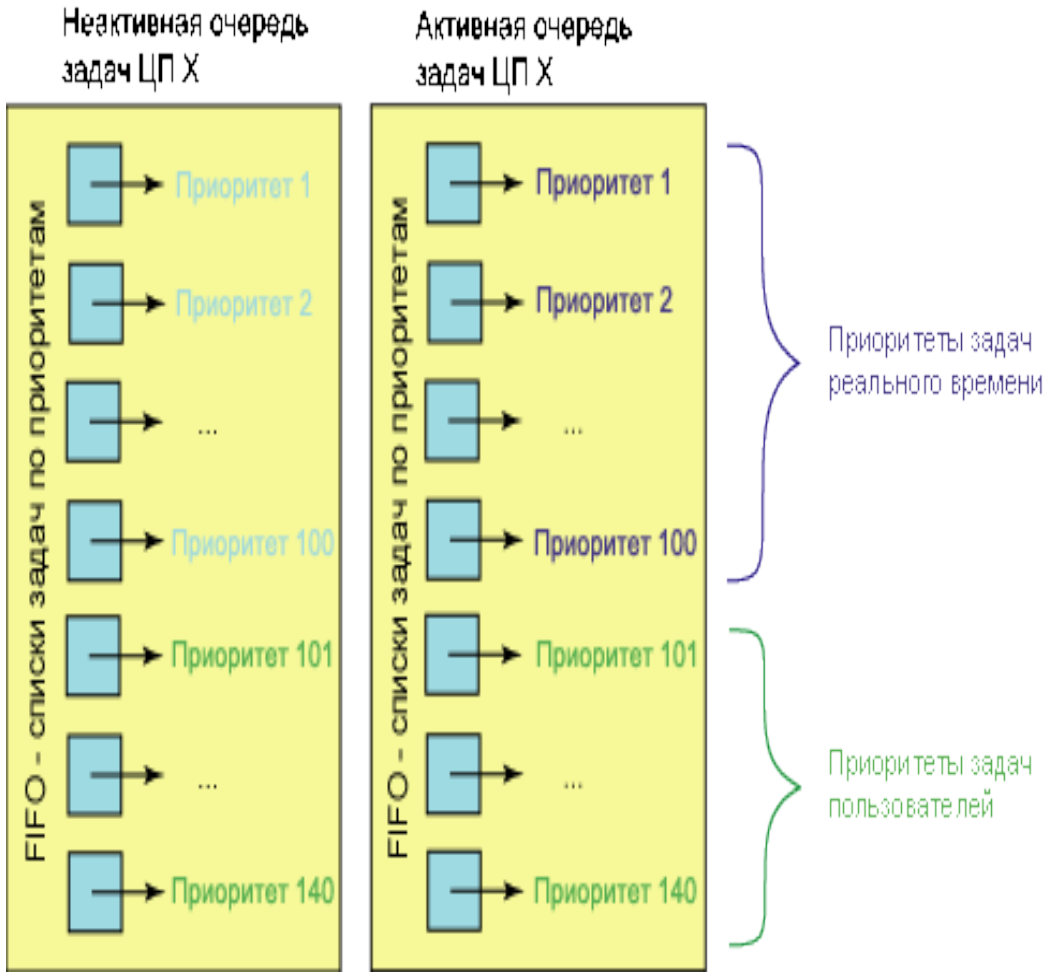


Определение задачи для запуска

Массив приоритетов длиной 140 бит



Бит 139
приоритет 139

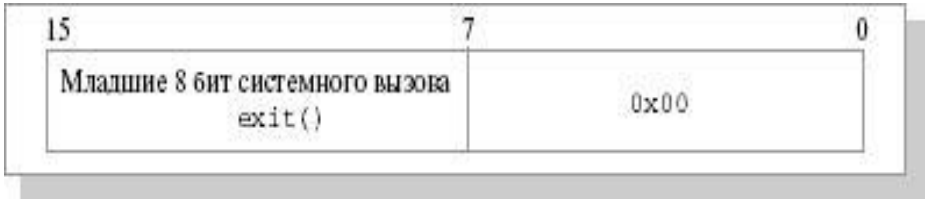


Синхронизация работы предков и потомков

pid_t wait(int *status); - приостанавливает выполнение текущего процесса до завершения какого-либо из его процессов-потомков и возвращает pid завершившегося процесса (-1 когда не имеет потомков).

status – если не NULL, то он указывает на переменную, в которую заносится состояние завершившегося процесса.

Если процесс завершился при помощи вызова функции exit()



Если процесс был завершен сигналом

Цикл ожидания завершения потомков

```
int pid1, pid2; /* идентификаторы процессов-потомков */
int status;/* статус завершения процесса-потомка */
int ret = 0;/* текущий возврат системного вызова wait */
while((ret = wait(&status)) != (-1)) {
/* обработка завершения 1-го потомка */
if(ret == pid1)
.....
/* обработка завершения 2-го потомка */
if(ret == pid2)
.....
} /*while */
```

Ожидание завершения процесса без цикла

pid_t waitpid(pid_t pid, int *status, int options);

Приостанавливает выполнение текущего процесса до завершения заданного процесса или проверяет завершение заданного процесса.

Если pid > 0, то он задает PID процесса, завершение которого ожидается.
Если pid = 0, то ожидает/проверяет завершение любого процесса той группы, к которой принадлежит текущий процесс.
Если pid < 0, то ожидает/проверяет завершение любого дочернего процесса, идентификатор группы процессов которого равен абсолютному значению *pid*.

options:

WNOHANG - не приостанавливать текущий процесс, если проверяемый процесс не завершился;

WUNTRACED - не приостанавливать текущий процесс также для потомков, которые завершились, но о состоянии которых еще не доложено;

0 - определяет переход в ожидание, если проверяемый процесс не завершился.

Возвращает:

Идентификатор дочернего процесса, который завершил выполнение.

-1 в случае ошибки или нуль, если использовался **WNOHANG**, но ни один дочерний процесс еще не завершил выполнение

Пример использование waitpid

```
#include <stdio.h>
#include <sys/wait.h>
int main() {
 int status1,status2;
 ipid_t child1,child2;
 int p1, p2, t;
 child1=fork(); /* Порождение первого потомка*/
 if (!child1)        { /* Операторы потомка 1 */
       sleep(10);  /* Задержка работы первого потомка  на 10 с */
       exit(10); /* Завершение работы первого потомка и передача кода завершения   предку*/
      }   else /* Предок */
   if (child1!=-1) {
     child2=fork();  /* Порождение второго потомка */
     if (!child2)  { /* Операторы потомка 2 */
       sleep(20); /* Задержка работы второго потомка на 20 с */
       exit(20); /* Завершение работы второго потомка и передача кода завершения   предку*/
      }
   else /* Предок */
        if (child2!=-1) {
           scanf("%d", &t);   /* ввод времени задержки предка*/
           sleep(t); /* Задержка работы предка*/
           p2=waitpid(child2,&status2,WNOHANG); /* проверка окончания работы   второго потомка */
           p1=waitpid(child1,&status1,WNOHANG); /* проверка окончания работы первого потомка */
           if (p1+p2==0) printf(" оба процесса не завершены \n ");
           else
              if (p1+p2==p1) printf("завершил работу первый потомок  \n");
             else
                if (p1+p2==p2) printf(" завершил работу второй потомок \n ");
                else   printf(" завершили работу оба потомка  \n");
       }       else       printf("Потомки не порождены\n");
    }       return(0);
 }
```

Задание 1 к лабораторной работе 4

Напишите программу, которая открывает текстовый файл, порождает поток, а затем ожидает его завершения. Потоку в качестве параметра передается дескриптор файла. Поток выводит на экран: класс планирования, текущий, минимальный и максимальный статические приоритеты, содержимое файла, а затем закрывает или не закрывает файл. После завершения работы потока, программа должна вывести свой текущий приоритет и проверить - закрыт ли файл, и если он не закрыт, то принудительно закрыть. Результат проверки должен быть выведен на экран.

Задание 2 к лабораторной работе 4

Напишите программу, которая открывает входной файл и два выходных файла. Затем она должна в цикле построчно читать входной файл и порождать два потока. Одному потоку передавать нечетную строку, а другому – четную. Оба потока должны работать параллельно. Каждый поток записывает в свой выходной файл полученную строку и завершает работу. Программа должна ожидать завершения работы каждого потока и повторять цикл порождения потоков и чтения строк входного файла, пока не прочтет последнюю строку, после чего закрыть все файлы.