Дополнение к отчету №3

Написать программу для завершения одного из потоков с клавиатуры:

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

pthread_t t1, t2;

void int_handler(int signum)
{
    printf("Caught SIGUSR1\n");
    pthread_exit(NULL);
}

void * thread_function1(void * arg)
{
    char c[100];
    int time = 0;
    while (1)
    {
      fgets (c, sizeof(c),stdin);
      if (c[0]=='q')
        {
        printf("Killing t2\n");
          pthread_kill(t2, SIGUSR1);
        }
    }
}

void * thread_function2(void * arg)
{
    int time = 0;
    while (1)
    {
        printf("Thread 2: %d\n", time++);
        sleep(1);
    }
}

int main()
{
    signal(SIGUSR1, int_handler);
    pthread_create(&t1, NULL, thread_function1, NULL);
    pthread_create(&t2, NULL, thread_function2, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}
```

Модифицировать программу, для завершения одного из потоков нажатием комбинации клавиш Ctrl+C

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


pthread_t t1, t2, t3;

void int_handler(int signum)
{
	printf("Caught Ctrl+C\n");
	pthread_cancel(t2);
}

void * thread_function2(void * arg)
{
	int time = 0;
	while (1)
	{
		printf("Thread 1: %d\n", time++);
		sleep(1);
	}
}

void * thread_function3(void * arg)
{
	int time = 0;
	while (1)
	{
		printf("Thread 2: %d\n", time++);
		sleep(1);
	}

}

int main()
{
	signal(SIGINT, int_handler);
	pthread_create(&t2, NULL, thread_function2, NULL);
	pthread_create(&t3, NULL, thread_function3, NULL);

	pthread_join(t2, NULL);
	pthread_join(t3, NULL);

	return 0;
}
```

Приоритеты процессов:

В BSD Unix значение приоритета является целым числом от -20 до +20: -20 относится к самым приоритетным процессам, а +20 - к процессам наименьшей приоритетности. Умалчиваемое значение приоритета (если не используется nice) равно 0. Точнее говоря, если пользователь не обращается к nice, приоритет запускаемого процесса будет унаследован от предка, в простейшем случае - оболочки, обычно имеющей нулевое значение приоритета. Рядовой пользователь может задавать только более высокие (положительные) значения приоритета, понижая приоритетность запускаемых процессов. Суперпользователь может задавать также отрицательные значения приоритета. Далее, если команду может выполнить обычный пользователь, в тексте будет указано приглашение "%", иначе - приглашение "#".

Диспетчеризация процессов:

На протяжении существования процесса выполнение его потоков может быть многократно прервано и продолжено.

Переход от выполнения одного потока к другому осуществляется в результате планирования и диспетчеризации. Работа по определению того, в какой момент необходимо прервать выполнение текущего активного потока и какому потоку предоставить возможность выполняться, называется планированием. Планирование потоков осуществляется на основе информации, хранящейся в описателях процессов и потоков. При планировании могут приниматься во внимание приоритет потоков, время их ожидания в очереди, накопленное время выполнения, интенсивность обращений к вводу-выводу и другие факторы. ОС планирует выполнение потоков независимо от того, принадлежат ли они одному или разным процессам. Так, например, после выполнения потока некоторого процесса ОС может выбрать для выполнения другой поток того же процесса или же назначить к выполнению поток другого процесса.

Планирование потоков, по существу, включает в себя решение двух задач:

- определение момента времени для смены текущего активного потока;
- выбор для выполнения потока из очереди готовых потоков.

В системе UNIX реализована вытесняющая многозадачность, основанная на использовании приоритетов и квантования.

Каждый процесс в зависимости от задачи, которую он решает, относится к одному из трех определенных в системе приоритетных классов: классу реального времени, классу системных процессов или классу процессов разделения времени. Назначение и обработка приоритетов выполняются для разных классов по-разному. Процессы системного класса, зарезервированные для ядра, используют стратегию фиксированных приоритетов. Уровень приоритета процессу назначается ядром и никогда не изменяется.

Процессы реального времени также используют стратегию фиксированных приоритетов, но пользователь может их изменять. Так как при наличии готовых к выполнению процессов

реального времени другие процессы не рассматриваются, то процессы реального времени надо тщательно проектировать, чтобы они не захватывали процессор на слишком долгое время. Характеристики планирования процессов реального времени включают две величины: уровень глобального приоритета и квант времени. Для каждого уровня приоритета по умолчанию имеется своя величина кванта времени. Процессу разрешается захватывать процессор на указанный квант времени, а по его истечении планировщик снимает процесс с выполнения.

Состав класса процессов разделения времени наиболее неопределенный и часто меняющийся в отличие от системных процессов и процессов реального времени. Для справедливого распределения времени процессора между процессами в этом классе используется стратегия динамических приоритетов. Величина приоритета, назначаемого процессам разделения времени, вычисляется пропорционально значениям двух составляющих: пользовательской части и системной части. Пользовательская часть приоритета может быть изменена администратором и владельцем процесса, но в последнем случае только в сторону его снижения.

Системная составляющая позволяет планировщику управлять процессами в зависимости от того, как долго они занимают процессор, не уходя в состояние ожидания. У тех процессов, которые потребляют большие периоды процессорного времени без ухода в состояние ожидания, приоритет снижается, а у тех процессов, которые часто уходят в состояние ожидания после короткого периода ' использования процессора, приоритет повышается. Таким образом, процессам, ведущим себя не «по-джентльменски», дается низкий приоритет. Это означает, что они реже выбираются для выполнения. Это ущемление в правах компенсируется тем, что процессам с низким приоритетом даются большие кванты времени, чем процессам с высокими приоритетами. Таким образом, хотя низкоприоритетный процесс и не работает так часто, как высокоприоритетный, но зато, когда он наконец выбирается для выполнения, ему отводится больше времени