Операционные системы
лекции 13, 14

- Взаимодействие процессов: надежные сигналы
- Взаимное исключение.
- Семафоры
- Мониторы

5.11.2010

```
int target_pid, cnt, fd[2], status;
void SigHndlr(int s)   {
      if (cnt < MAX)    {
            read(fd[0], &cnt, sizeof(int));printf("%d \n", cnt); cnt++;
            write(fd[1], &cnt, sizeof(int));
            kill(target_pid, SIGUSR1);
      }
      else if (target_pid == getppid()) {
         close(fd[1]); close(fd[0]); exit(0);
              } else
                   kill(target_pid, SIGUSR1);
}
int main(int argc, char **argv){
      pipe(fd);  signal (SIGUSR1, SigHndlr);
      if (target_pid = fork())    { write(fd[1],&cnt, sizeof(int));
     wait(&status);close(fd[1]); close(fd[0]);return 0;
      }
      else{ read(fd[0],&cnt,sizeof(int));
    target_pid = getppid();     write(fd[1],&cnt,sizeof(int));
            kill(target_pid, SIGUSR1);
             while(1) sleep(5);
      }
}  /* программа с исправлениями*/
```

Использование надежных сигналов

Нужно:

- возможность блокировать сигналы (на время выполнения обработчика)

- возможность указать, сбрасывать ли обработку, на обработку по умолчанию

- перезапускать ли сист. вызовы при возврате из обработчика?

Множество сигналов

<signal.h>

тип sigset_t используется для хранения множества сигналов

```
int sigemptyset(sigset_t *pset);
```

очищает множество сигналов, на которое указывает pset
множество становится пустым

```
int sigfillset(sigset_t *pset);
```

заполняет множество:включает в него все сигналы

```
int sigaddset(sigset_t *pset, int signum);
```

добавить сигнал signum в множество

```
int sigdelset(sigset_t *pset, int signum);
```

удалить сигнал signum из множества

```
int sigismember(const sigset_t *pset, int signum);
```

проверка наличия сигнала в множестве

Работа с маской сигналов

Маска — множество заблокированных в текущий момент времени сигналов.
Каждый процесс имеет свою маску сигналов, она наследуется потомком при fork().

```
int sigprocmask(int mode,const sigset_t *pset,
                                sigset_t *poldset);
```

SIG_BLOCK
SIG_UNBLOCK
SIG_SETMASK

Установка обработчика

```
int sigaction(int signum, const struct sigaction *act,
                          struct sigaction *oldact);
struct sigaction {
  void (*sa_handler)(int);
  void (*sa_sigaction)(int, siginfo_t *, void *);
  sigset_t sa_mask;
  int sa_flags;
  void (*sa_restorer)(void);
};
-------------------------
Пример:   void f (int s){/*обработчик*/}
/*Установка*/
struct sigaction old, new;
new.sa_handler = f;
new.sa_flags=SA_RESTART;
sigemptyset(&new.sa_mask);/*никакие сигналы не блокир.*/
sigaction(SIGINT,&new,&old);
...
sigaction(SIGINT,&old,NULL);/*восст. предыд. обработчик*/
```

Работа с разделяемыми ресурсами

Параллельные процессы
-независимые
-взаимодействующие

Критический ресурс — доступен в текущий момент времени только одному процессу (внешнее устройство, разделяемая переменная)

Задачи ОС:
1. Распределение ресурсов
2. Организация защиты ресурса от несанкционированного доступа других процессов

Организация взаимного исключения

Взаимное исключение — это способ работы с разделяемым ресурсом, при котором пока один процесс работает с ресурсом, другие не имеют к ресурсу доступа.

Критическая секция

Пример: сторож и тигр

Проблемы при организации взаимного исключения:

1. **Возникновение тупиков (deadlocks).**
2. **Бесконечное (долгое) ожидание ресурса процессом**

Правила организации взаимного исключения:

1. Процесс, находящийся *вне* своей критической секции, не должен блокировать выполнение другого процесса.
2. Порядок попадания процессов на процессор не должен влиять на результат программы

Способы организации взаимного исключения

1. Запретить все прерывания непосредственно перед входом в критическую секцию.

Что будет дальше???

2.Переменная блокировки.
Тот, кто занимает ресурс может это сделать, если в переменной 0. Он устанавливает ее в 1. Когда освобождает ресурс, присваивает в переменную блокировки 0.

Проблема использования?

Способы организации взаимного исключения

Алгоритм Петерсона (1981) **Пример для двух процессов**

```
#define FALSE 0
#define TRUE 1
#define N 2
int turn;
int interested[N];

void enter(int process){
  int other =1- process;
  interested[process]=TRUE;
  turn = process;
  while(turn==process && interested[other]==TRUE) ;
}

void leave(int process){
  interested[process]=FALSE;
}
```

Семафоры

Дейкстра (1965)

Семафор - целая переменная, принимающая только неотрицательные значения, над которой определены две атомарные операции:

DOWN — если значение семафора >0, операция уменьшает его на 1 и возвращает управление, если значение семафора =0, DOWN не возвращает управление, а процесс переводится в состояние ожидания.

UP – увеличивает значение семафора на 1. Если с ним были связаны один или несколько ожидающих завершения операции DOWN процессов, то один из них разблокируется и завершает свою операцию DOWN.

Семафоры

Реализация взаимного исключения:

Процесс 1

```
down(S);
/*критическая секция */
up(S);
```

Процесс 2

```
down(S);
/*критическая секция */
up(S);
```

Мониторы

Монитор - совокупность переменных и процедур.
Организована таким образом, что
– в каждый момент времени может выполняться не более одной процедуры манипулирующей этими переменными.
– Структуры данных, входящие в монитор, доступны только для процедур этого монитора.
– Процесс входит в монитор, путем вызова одной из его процедур
– В любой момент времени внутри монитора может находиться не более одного процесса, выполняющего процедуры. (Если процесс пытается попасть в монитор, в котором другой процесс – он блокируется).

Идея монитора — Хоар (1974)

Реализация монитора

/*используем семафоры */

```
struct Account {
  semaphore lock;
  int amount;
};

void deposit(struct Account *a, int m) {
  down(&a->lock);
  a->amount += m;
  up(&a->lock);
}

void take(struct Account *a, int m) {
  down(&a->lock);
  a->amount -= m;
  up(&a->lock);
}
```