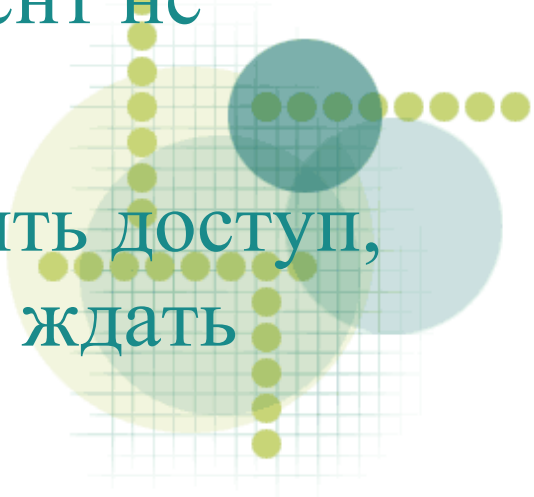
Классические задачи синхронизации, ч. 2

- Читатели и писатели
- Производители и потребители
- Спящий парикмахер

Читатели и писатели

- Дана некоторая разделяемая область память
- К этой структуре данных может обращаться произвольное количество «читателей» и произвольное количество «писателей»
- Несколько читателей могут получить доступ одновременно, писатели в этот момент не допускаются
- Только один писатель может получить доступ, другие писатели и читатели должны ждать



Решение 1

- Первое решение: читатель может войти в критическую секцию, если нет писателей
- Это решение несправедливо, так как отдает предпочтение читателям
- Плотный поток запросов от читателей может привести к тому, что писатель никогда не получит доступа к критической секции: ситуация «голодания» (starvation)

Решение 2

- Отдадим предпочтение писателям, то есть читатель не входит в критическую секцию, если есть хотя бы один ожидающий писатель

```
pthread_mutex_t m;
pthread_cond_t cw, cr;
int rcnt, wcnt;
int wwcnt; // число ожидающих писателей
void rdlock()
{
  pthread_mutex_lock(&m);
  while (wcnt > 0 || wwcnt > 0)
    pthread_cont_wait(&cr, &m);
  rcnt++;
  pthread_mutex_unlock(&m);
}
```

Решение 2

```
void wrlock()
{
  pthread_mutex_lock(&m);
  while (wcnt > 0 || rcnt > 0) {
    wwcnt++;
    pthread_cond_wait(&cw, &m);
    wwcnt--;
  }
  wcnt++;
  pthread_mutex_unlock(&m);
}

void unlock()
{
  // …
}
```

Решение 2

- Данное решение отдает приоритет писателям, и тоже несправедливо
- Возможно «голодание» (starvation) читателей
- Третье решение: не отдавать никому приоритета, просто использовать мьютекс

Производители-потребители (producer-consumer problem)

- Дан буфер фиксированного размера (N), в котором размещается очередь.
- Производители добавляют элементы в конец очереди, если буфер заполнился, производители засыпают
- Потребители забирают элементы из начала очереди, если буфер пуст, потребители засыпают

Производители-потребители

```
int buf[N];
int head, tail;
pthread_mutex_t m;
pthread_cond_t cc; // consumer condvar
pthread_cond_t pc; // producer condvar
void put(int x)
{
  pthread_mutex_lock(&m);
  while ((tail + 1) % N == head)
    pthread_cond_wait(&pc, &m);
  buf[tail] = x;
  tail = (tail + 1) % N;
  if ((head + 1) % N == tail)
    pthread_cond_signal(&cc);
  pthread_mutex_unlock(&m);
}
```

Производители-потребители

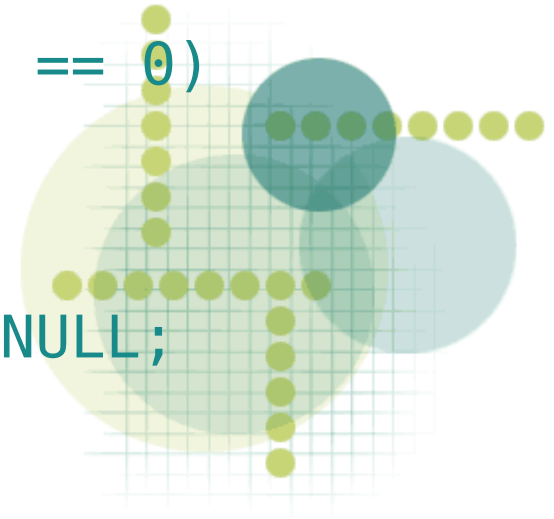
```
int get(void)
{
  int val;
  pthread_mutex_lock(&m);
  while (head == tail)
    pthread_cond_wait(&cc, &m);
  val = buf[head];
  if ((tail + 1) % N == head)
    pthread_cond_signal(&pc);
  head = (head + 1) % N;
  pthread_mutex_unlock(&m);
  return val;
}
```

Спящий парикмахер (sleeping barber)

- В парикмахерской имеется одно кресло для стрижки и N кресел для ожидающих посетителей
- Если нет посетителей, парикмахер спит
- Если приходит посетитель и кресло для стрижки свободно, посетитель садится в него и парикмахер начинает его стричь
- В противном случае посетитель садится в кресло для ожидающих
- Если все кресла заняты, посетитель уходит

Спящий парикмахер

```
pthread_mutex_t m;
pthread_t chair_thr;  // кого стрижем
int wait_cnt;   // сколько посетителей ожидают
pthread_cond_t bc; // barber condvar
pthread_cond_t cc; // consumer condvar
void barber(void)
{
  while (1) {
    pthread_mutex_lock(&m);
    while (chair_thr == NULL && wait_cnt == 0)
      pthread_cond_wait(&bc, &m);
    pthread_mutex_unlock(&m);
    make_haircut();
    pthread_mutex_lock(&m); chair_thr = NULL;
    pthread_cond_signal(&cc);
    pthread_mutex_unlock(&m);
  }}
```



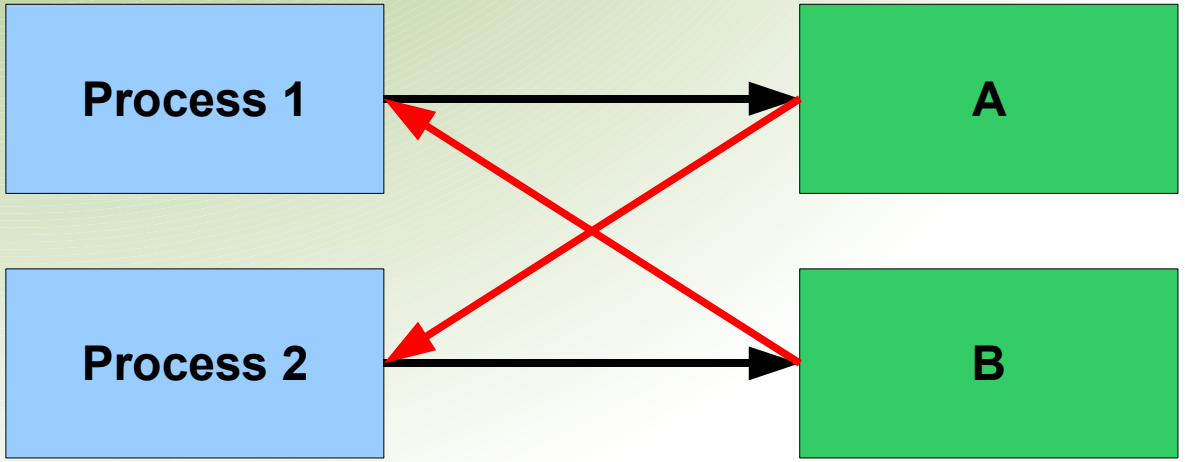
Спящий парикмахер

```
int consumer(void) {
  pthread_mutex_lock(&m);
  if (chair_thr != NULL && wait_cnt == N) {
    // no space, leaving
    pthread_mutex_unlock(&m);
    return -1;
  }
  while (chair_thr != NULL) {
    wait_cnt++;
    pthread_cond_wait(&cc, &m);
    wait_cnt--;
  }
  chair_thr = pthread_self();
  pthread_cond_signal(&bc);
  pthread_mutex_unlock(&m);
  get_haircut(); return 0;
}
```

Обнаружение тупиков

```
Process 1:
lock(&A);
lock(&B);
```

```
Process 2:
lock(&B);
lock(&A);
```



- Захваченный ресурс — дуга от процесса к ресурсу
- Ожидаемый ресурс — дуга от ресурса к процессу
- Если в графе есть цикл, система попала в состояние тупика

Группы процессов

- Группа процессов — процессы, объединенные для выполнения задачи (например, для выполнения конвейера)
- Группа процессов выступает как единое целое при
  - Получении сигналов, в особенности, от терминала (например, Ctrl-C — SIGINT)
  - При работе с терминалом (основная и фоновые группы процессов)
- Идентификатор группы процессов — это идентификатор одного из процессов в группе

Создание группы

```
#include <unistd.h>
pid_t getpgid(pid_t pid);
int setpgid(pid_t pid, pid_t pgid);
```

- Группу процессов можно получить только у процесса из текущей сессии, при этом если pid == 0, возвращается группа процессов текущего процесса
- Для setpgid pid == 0 означает текущий процесс, pgid == 0 — группа процессов с pgid текущего процесса

Создание группы, особые случаи

```
setpgid(0, 0);
```

- Процесс создает новую группу процессов и помещает в нее себя (выполняется в сыне)

```
setpgid(0, pgid);
```

- Процесс помещает себя в существующую группу процессов (в сыне)

```
setpgid(pid, pid);
```

- Процесс создает новую группу процессов и помещает туда указанный процесс (в отце)

Группы процессов и терминал

- У терминала может быть одна основная группа процессов и произвольное количество фоновых групп процессов
- Основная группа процессов:
  - Имеет право чтения с терминала (попытка чтения для фоновой группы процессов вызывает приостановку процесса фоновой группы)
  - Получает сигналы SIGINT, SIGQUIT с терминала

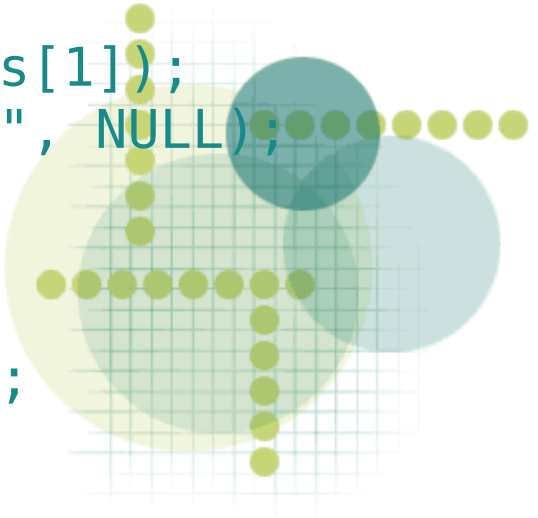
Основная группа процессов терминала

```
pid_t tcgetpgrp(int fd);
int tcsetpgrp(int fd, pid_t pgrp);
```

- fd — любой файловый дескриптор терминала (например, 0 — стандартный ввод)
- tcsetpgrp устанавливает основную группу процессов терминала

Пример: ls -l | wc -l

```
int main(void) {
  pipe(fds);
  if (!(pid1 = fork())) {
    setpgid(0, 0); tcsetpgrp(0, getpid());
    dup2(fds[1], 1); close(fds[0]); close(fds[1]);
    execlp("/bin/ls", "/bin/ls", "-l", NULL);
  }
  setpgid(pid1, pid1); tcsetpgrp(0, pid1);
  if (!(pid2 = fork())) {
    setpgid(0, pid1);
    dup2(fds[0], 0); close(fds[0]); close(fds[1]);
    execlp("/usr/bin/wc", "/usr/bin/wc", "-l", NULL);
  }
  setpgid(pid2, pid1);
  close(fds[0]); close(fds[1]);
  wait(0); wait(0); tcsetpgrp(0, getpgid(0));
  return 0;
}
```



Процессы-демоны

Планирование процессов

- Планировщик — компонента ядра операционной системы
- Планировщик определяет, какой процесс из числа готовых к выполнению назначается на выполнение на ЦП
- Типы планировщиков:
  - Пакетный
  - Разделения времени
  - Реального времени

Пакетное планирование

- Цель — обеспечить максимальную пропускную способность ВС (то есть максимальное число выполненных задач)
- Ядро переключается с одного на другой процесс при следующих условиях:
    - Выполнявшийся процесс завершил работу
    - При выполнении возникла фатальная ошибка или процесс исчерпал отведенные ему ресурсы
    - Выполнявший процесс инициировал операцию, которая не может быть выполнена немедленно
    - Процесс запросил добровольное переключение

Планирование разделения времени

- Цель: разделить процессорное время между процессами, готовыми к выполнению
- Ядро переключается с одного процесса на другой при следующих условиях
  - Процесс завершил работу
  - При выполнении возникла ошибка
  - Процесс инициировал операцию, которая не может быть выполнена немедленно
  - Истек квант времени выполнения процесса
  - Процесс запросил добровольное переключение

Классификация процессов

- По поведению
  - «I/O-bound» - процесс выполняет активный обмен с внешними устройствами и проводит много времени в ожидании ввода-вывода (пример: веб-сервер, редактор текста)
  - «CPU-bound» - процессы, интенсивно занимающие процессорное время (пример: компиляция программ, вычислительные задачи, рендеринг изображений и т. п.)

Классификация процессов

- По назначению:
  - Интерактивные — основное время проводят в ожидании пользовательского ввода, при поступлении ввода должны быстро активироваться, чтобы не было ощущения «торможения»
  - Пакетные — не ожидают ввода пользователя (компиляторы, численные приложения...)

Параметры планирования процессов разделения времени

- Значение nice: [-20, 19] — чем меньше значение, тем выше приоритет. 0 — приоритет по умолчанию
- Приоритет группы процессов: grpnice
- Приоритет пользователя: usrnice
- Полный приоритет: nice + grpnice + usrnice отсеченное по интервалу [-20; 19]
- Нормализованный приоритет normprio = 20 - fullnice, находится в интервале [1, 40] — чем больше значение, тем больше приоритет

Планирование в Linux

- Планирование процессов разделено на эпохи
- В начале каждой эпохи каждому процессу назначается базовый квант
  base_quantum = normprio/4 + 1
- counter — число «неотработанных» квантов в эпохе, изначально counter = base_quantum
- За каждый квант, когда процесс выполняется, значение counter уменьшается на 1
- Приоритет: priority = counter + normprio
- Выбирается процесс с наибольшим приоритетом

Планирование в Linux

- Эпоха заканчивается, когда у всех готовых к выполнению процессов counter == 0
- В начале очередной эпохи:
  base_quantum = counter/2 + normprio/4 + 1
- Таким образом приоритет отдается I/O-bound процессам

Планирование реального времени

- Цель: обеспечить минимальное время отклика, то есть время от наступления события до постановки на выполнение процесса, ожидающего этого события
- Виды планирования реального времени
  - На основе фиксированного расписания
  - На основе статических приоритетов

Планирование реального времени

- Ядро переключается с одного процесса на другой при следующих условиях
  - Процесс завершил работу
  - При выполнении возникла ошибка
  - Процесс инициировал операцию, которая не может быть выполнена немедленно
  - Готов к выполнению процесс с большим приоритетом
  - Истек квант времени выполнения процесса
  - Процесс запросил добровольное переключение

Статический приоритет

- Каждый процесс реального времени имеет статический приоритет [1;99]
- Процессы разделения времени имеют статический приоритет 0, то есть назначаются на выполнение, только если нет готовых к выполнению процессов реального времени

Типы планирования р.в.

- SCHED_FIFO — нет квантования времени, процесс выполняется, пока не появится более высокоприоритетный процесс, либо процесс не начнет ввод-вывод, либо не будет снят
- SCHED_RR (round-robin) — выполнение квантуется, процессы одного приоритета выполняются по очереди

Инверсия приоритета (priority inversion)

- Предположим, что низкоприоритетный процесс P1 захватил некоторый ресурс R
- В это время стал готов к выполнению высокоприоритетный процесс P2, которому требуется ресурс R. Процесс P2 ожидает освобождения ресурса R процессом P1
- В это время может быть назначен на выполнение среднеприоритетный процесс P3, который еще более отсрочит время освобождения ресурса R процессом P1

Инверсия приоритета

- Проблема возникает, потому что процессы, ожидающие освобождения ресурса неявно получают приоритет процесса, захватившего ресурс.
- Отсрочка выполнения высокоприоритетного процесса может иметь катастрофические последствия
- Однозначного решения проблемы не существует
- Возможный вариант: назначать процессу, захватившему ресурс, максимальный приоритет ожидающего процесса (наследование приоритета)

Управление приоритетами в Linux

```
int nice(int inc);
void sched_yield(void);
int sched_setscheduler(pid_t pid, int policy,
                       const struct sched_param *param);
```