Динамическая память

Работа с памятью

Системные вызовы (man 2 …):

- brk/sbrk — устаревшие
- mmap — современный

Системный вызов — дорого и негибко, функции (man 3 …):

- Выделение:
  - malloc/calloc/realloc
- Освобождение:
  - free

Раз библиотечные функции, то возможна своя реализация:

- dlmalloc (GNU), phkmalloc (BSD)
- jemalloc (BSD, Firefox, Facebook), tcmalloc (Google), ptmalloc (GNU)

Зачем новая библиотека

Чтобы понять причины создания новых менеджеров памяти требуется понимание:

- страничной организации памяти, виртуальной памяти, защиты памяти
- работы кеша и памяти
- особенности многопоточных приложений
- особенности современного выделения памяти при fork
- особенности завершения работы программы.
- общая эффективность работы программ

Страничная организация памяти

Пока нам важно знать, что:

- страница — это минимальный объект, на который распространяются общие свойства. В частности, меньше, чем страницу, мы программе памяти выделить не можем (она может запросить 100байт, но выделим мы не меньше 4KB)
- Страница может быть помечена как RO (только для чтения), так помечаются:
  - Выполнимые (с кодом) страницы, поэтому процесс заменить свой код не может.
  - Память с константами (все константы из программы).
- Если процесс попробует сделать что-то не то со страницей (пойдет не в свою память, будет писать в память, в которую писать нельзя), то процессор эту операцию делать не будет, а через механизм прерываний передаст управление ОС, которая и примет решение, что делать дальше (через выполнение своих команд на процессоре).
- Виртуальная страница: все адреса являются виртуальными, с помощью отдельного механизма отображаемыми аппаратно на физические (правило преобразования задает ОС).
- Все флаги страниц, TLB — пока знать не нужно.

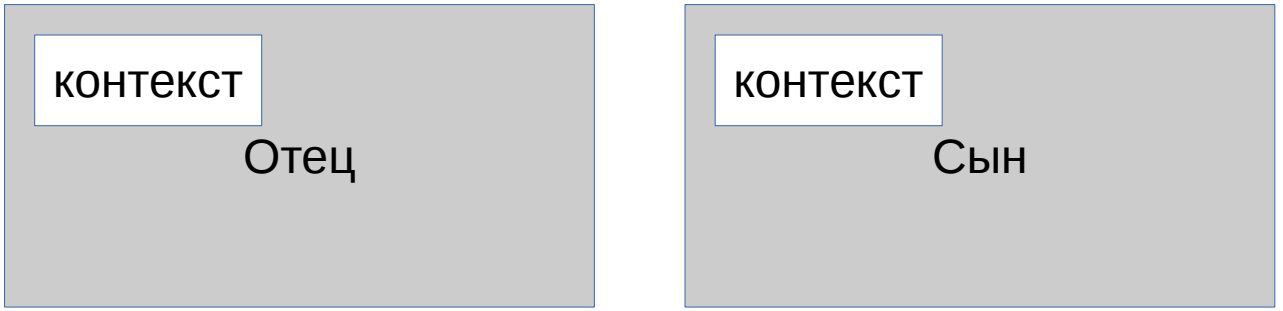
Работа кеша и памяти

- Кеш расположен близко к процессору и служит «шлюзом» между ним и оперативной памятью
- При запросе данных из памяти в кеш попадают не только эти данные, но и соседние (сколько справа и слева — зависит от алгоритмов кеша).
- За счет этого при запросе соседней ячейки она будет взята из кеша, без обращения к шине данных
  - L1/L2 кеши, связь с TLB рассмотрим позже.

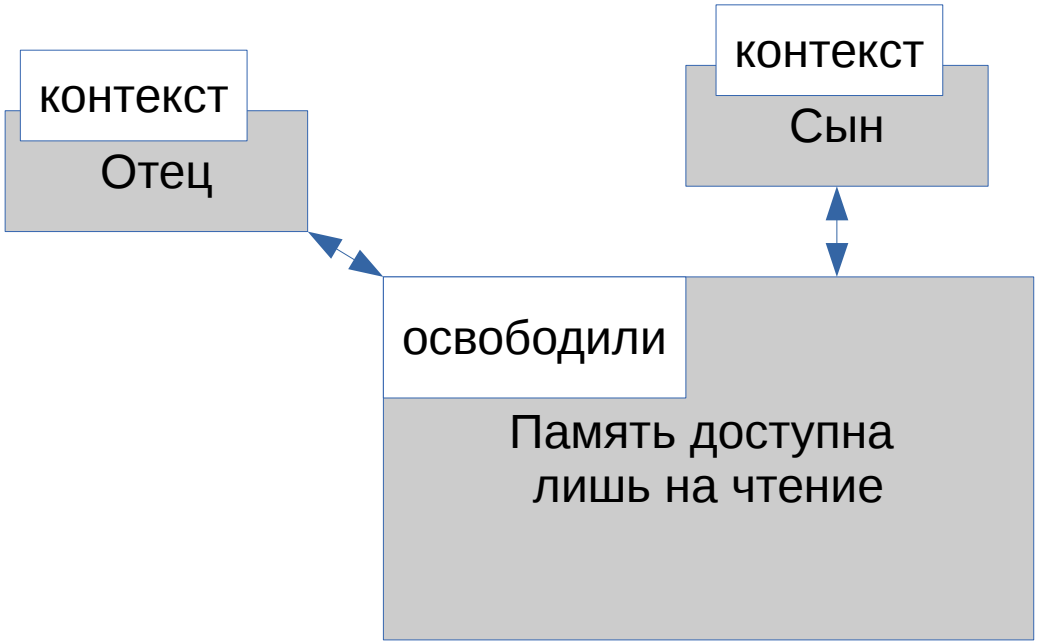
Многопоточные приложения

- Общая память, к которой могут обращаться разные нити одновременно
- Механизм раздельного допуска — мьютекс/семафор (рассмотрим при изучении IPC), пока достаточно понимать, что возникнет очередь из нитей на доступ к общим переменным

Память при fork



Там нам показывают, но часто делают fork + execve, а зачем копировать память (делать независимую копию), если потом освобождать?



- Давайте вместо этого память пометим как RO.
- Если отец и сын память менять не будут, то всё хорошо.
- Если один из них скажет _exit/execve, то тоже нормально, снимем со страниц этот бит.
- Если отец/сын начнут менять память, то ОС именно эту страницу и скопирует, после чего изменит, пусть отцу, соответствие между виртуальной и физической страницей.

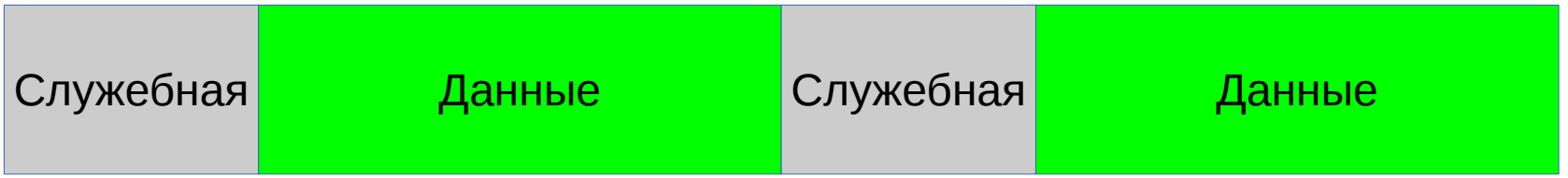
Общая эффективность выполнения

Эффективный — тот что выполнится быстрее на нашем оборудовании. Есть инженерные решения, которые могут быть неэффективными при «неправильном» использовании:

- Многопроцессорность:
  - Если несколько потоков меняют одни и те же данные, то из-за синхронизации кеша общая скорость падает (ждут)
- Конвейер:
  - спагетти-код (постоянные и непредсказуемые переходы его ломают)
- Оперативная память:
  - Слишком много страниц, частая смена адресов.

Старый malloc

- Минимальный расход памяти, все области расположены рядом
- Тогда многопроцессные/многоядерные системы не были распространены и многопоточных приложений не было (почти не было).



Проблема с fork

Проблема при fork+execve/_exit:

- Много мелких объектов и fork:
  - изначально новые страницы не выделяются (почти)
  - сын запускает освобождение объектов (destroy/free/atexit)
  - начинают меняться служебные области объектов — перекопирование страниц
  - ОС осуществила копирование многих страниц
  - Сын сказал, что ему вся эта память не нужна

Еще проблемы

- Проблемы с фрагментацией:
  - взяли несколько мелких объектов, потом решили объект освободить/увеличить — дырка, которую тяжело найти и использовать, т. к. надо просмотреть всю память.
- Проблемы с многопроцессорностью:
  - у каждого процессора свой кеш, если память меняется в обоих, то надо синхронизировать.
- Проблема с многопоточностью:
  - Несколько нитей вызвали malloc — встанут в очередь, даже если каждая нить сидит на своем процессоре.

Jemalloc

- Несколько Арен (если процессоров больше одного), по 4 арены на процессор
- Служебная информация и данные лежат в разных страницах
  - процесс не может испортить служебную область!
- Битовая маска занятых мелких объектов и красное дерево для больших объектов — быстрый способ найти свободное место
- Нити на разных процессорах используют разные Арены — кеширование Арен не конфликтует между собой, изменения в Аренах можно делать параллельно (без семафоров).

Tcmalloc и ptmalloc

Принципы схожи. Ptmalloc — развитие dlmalloc.

По сравнению с jemalloc tcmalloc:

- чуть эффективнее, если нити, использующие malloc, часто порождаются/умирают
- чуть хуже, если нити долго живут

Какие проблемы видите?

```
while (i<N){
  i++;
  s=realloc(s,i);
  if(!s){
    perror(NULL);
    exit(1);
  }
}
```

```
while (i<N){
  I*=2;
  s=realloc(s,i);
  if(!s){
    perror(NULL);
    exit(1);
  }
}
```

Стратегии в malloc

```
#include<stdio.h>
#include <stdlib.h>

int main(){
  char * p,*s=malloc(1);
  int i=1;
  p=s;
  for(i=2;i<1000000;i++){
    s=realloc(s,i);
    if(p!=s){
      printf("%d %ld\n",i,s-p);
      p=s;
    }
  }
}
```

Результаты:

- Jemalloc:
  8 4096
  16 45056
  32 4096
  48 12288
  64 4096
  …
  переключения между областями до 14336, дальше переключение на 655360
- Ptmalloc:
  135128 139848032366592
  135144 -139264
  274408 -278528
  552936 -557056

Сценарии предоставления памяти самой ОС

- Доступной оперативной памяти: 1GB
- Приложение П1, при старте 10MB, потом запрашивает +500 MB
- Приложение П2: аналогичное

Сценарий 1: нет подкачки, обычное поведение

- Запустилось П1, запустилось П2.
- П1 получило +500 MB
- П2 получило NULL

Сценарий 2: есть подкачка (1GB), обычное поведение

- Запустилось П1, запустилось П2.
- П1 получило +500 MB
- П2 получило +500 MB
- Когда/если активно-используемых страниц станет не хватать, происходит обмен между оперативной памятью и подкачкой

Сценарий 3: нет подкачки, overcommit (Linux)

- Запустилось П1, запустилось П2.
- П1 получило +500 MB
- П2 получило +500 MB
- Когда/если активно-используемых страниц станет не хватать, происходит «убийцы» OOMKiller, ресурсы (освободившаяся оперативная память) распределяется между оставшимися.

Пример на fork

```
#include <iostream>
#include <unistd.h>

using namespace std;

//Что (сколько) будет выведено?

int main() {
    cout << "A";
    if (fork()) cout << "AA";
}
```

«Неожиданный» результат, т. к.:

- Системный вызов — дорог и негибок не зависимо от языка (свойство ОС)
- Выгодно кешировать — использовать память
- Память у сына дублируется