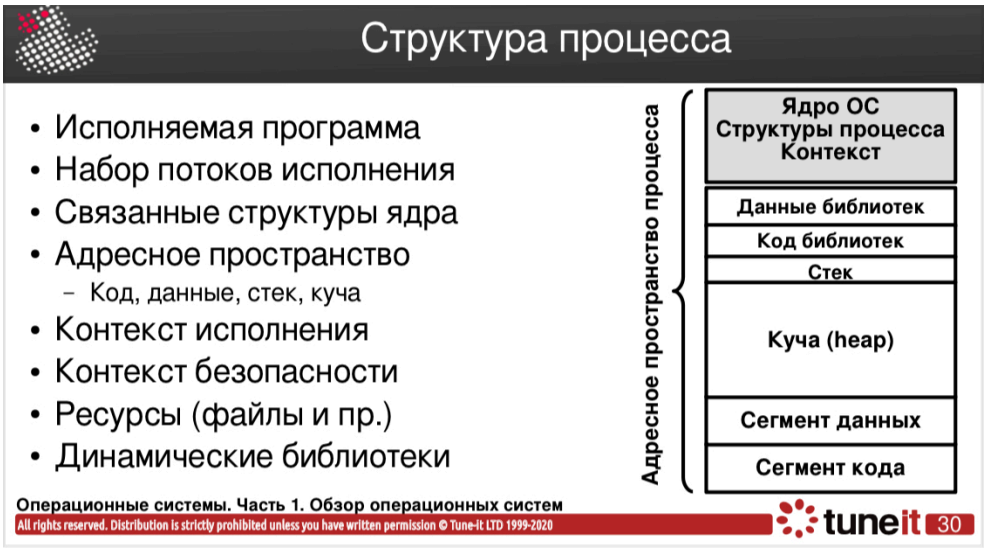
ОСИ Лаба1 Защита

1. Структура процесса

Структура процесса



Раньше потоков вообще не было, были тупо процессы (интересно наблюдать, как люди в старые времена справлялись без потоков)

Чем потоки лучше процессов?
Они легче весят. Чтобы создать процесс, нужно создать дохрена всего, а именно, адресное пространство (смотри выше)
А для потоков можно тупо создать доп набор регистров в памяти.
Поэтому, если нужно быстро создать счётную задачу, которая будет выполняться на отдельном процессоре, мы созданием потока экономим кучу времени, чтобы не создавать доп структуры памяти(хотя кроме памяти там ещё огого создаётся)

Связанные структуры ядра:
Когда процесс создаётся внутри ОС, ядро должно на все те ресурсы, которыми процесс будет оперировать, построить необходимое количество структур (областей памяти) происходит это внутри самого ядра. Все эти структуры отвечают за описание всех тех ресурсов, которые процесс содержит

В Линукс структура taskStruct

Каждая абстракция, которая связана с нашим процессом (pull of threads, сегментом памяти, подключённым библиотекам) всегда

внутри ядра существуют связанные с этими ресурсами структуры, которые описывают их и позволяют ядру ими манипулировать.

Сегмент кода хранит машинные команды. Характеристика сегмента - можно читать, выполнять, нельзя изменять

Сегмент данных - нельзя исполнять, можно менять. Есть один нюанс: copy on write

Куча - память, которая выделятся динамически

Стек - адреса возвратов + параметры передаваемых функций

Сегменты для разделяемых библиотек - данные библиотек, код библиотек
libc
Зачем? Чтобы не делать образы процессов излишне большими
Не очень хочется компилить и собирать каждый раз программу статически слинкованную (все функции которые

Можно прилинковать библиотеку статически, а можно динамически,

Выше опред границы в адресном пространстве - отмапленное ядро
При переключении контекста - уровень пользователя меняется на уровень ядра
После завершения выполнения структура процесса пропадает из памяти ?

Прерывания как раз и нужны чтобы уйти в контекст ОС при выполнении любой системной функции

Контекст исполнения - многозначное понятие. Например, набор регистров, которые содержат состояние нашей программы и в то же время это может быть уровень привилегий (уровень пользователя, уровень ядра)

Интел до 4 колец безопасности

Контекст безопасности - различные идентификаторы. Идентификатор пользователя, группы, suid (пользовательский id меняется на рута)
Контекст безопасности как раз и предназначен для того, чтобы понимать , от имени какого пользователя запущен процесс

Ресурсы - (ссылки на ресурсы = открытые файлы, сетевые

соединения,

Динамические библиотеки - файл, содержащий машинный код. Загружается в память **процесса** загрузчиком программ операционной системы либо при создании **процесса**, либо по запросу уже работающего **процесса**, то есть динамически. ... **Библиотеки**, используемые одной программой и содержащие дополнительные функции.

Task struct (все страшно, вся информация о процессе)

В нормальных линуксах - proc и user

2. **Виртуальная память**

Управление памятью

- Изоляция процессов
- Управление выделением и освобождением памяти (есть аллокаторы, в Java - new, heap allocator, kernel allocator, mapping files - функция nmap. Единственная область применения: создаёт регион памяти)
- Поддержка модулей (динамическая загрузка модулей. Модуль - драйвер, программа. Все ядра разбиты на большое количество модулей. Мы грузим модули, нам нужно выделить память далее - выгрузить, память удалить. Поддержка этих модулей- сложнаааа. Ядро ОС тоже состоит из нескольких сегментов. Когда мы в программу пытаемся загрузить динамически,
- Защита и контроль доступа (права на сегменты памяти. Буффера обычно находятся в стеке. Хакеры переполняло бцффер, клали злонамеренный код в стек и его исполняли -> люди разрабатывающие ОС стали добавлять доп битики, чтобы защитить исполнение кода внутри стека, внутри данных
- Долговременное хранение (сделать так, чтобы то, что мы поменяли в памяти отражалось на диске)
- Страничный обмен (paging, swapping ????????

Виртуальная память

- Отдельное виртуальное адресное пространство для каждого процесса и ядра. Внутри АП су
- Использование подкачки страниц с диска для эффективного использования памяти (если страницу не меняем - ее можно выгрузить на диск (если до этого была изменена)
- Управление MMU и TLB - memory management unit - **на уровне ОС**

создаёт и проверяет мэппинги наличия страниц в памяти. TLB - большой кэш, который позволяет кэшировать. Кэширует обращения, чтобы каждый раз не лазать по MMU
- Невыгружаемые страницы (страница очень часто используется, ее выгружать в swap не надо.

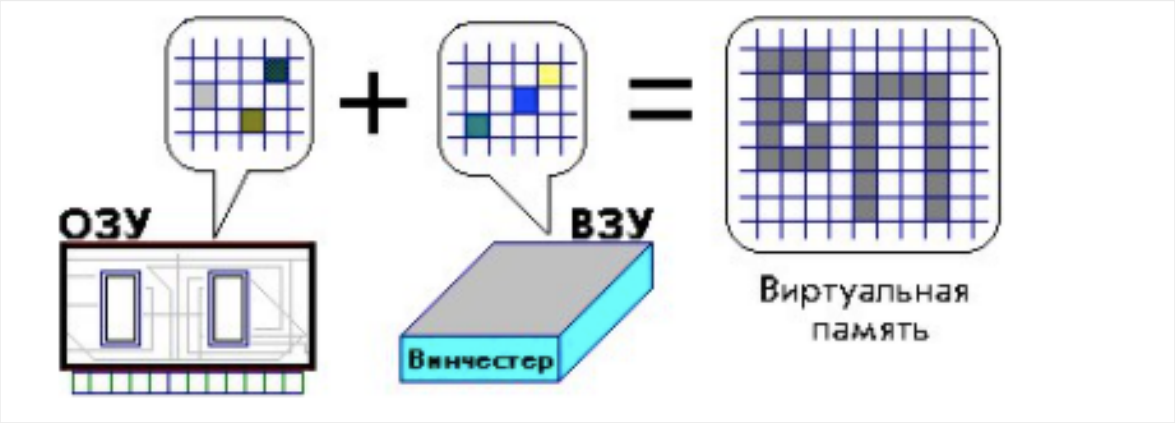
Виртуальная память — метод управления памятью, которая реализуется с использованием аппаратного и программного обеспечения компьютера. Она отображает используемые программами виртуальные адреса в физические адреса в памяти компьютера. Основная память представляется в виде непрерывного адресного пространства или набора смежных непрерывных сегментов. Операционная система осуществляет управление виртуальными адресными пространствами и соотнесением оперативной памяти с виртуальной. Программное обеспечение в операционной системе может расширить эти возможности, чтобы обеспечить виртуальное адресное пространство, которое может превысить объем оперативной памяти и таким образом иметь больше памяти, чем есть в компьютере. Виртуальная память позволяет модифицировать ресурсы памяти, сделать объём оперативной памяти намного больше, для того чтобы пользователь, поместив туда как можно больше программ, реально сэкономил время и повысил эффективность своего труда. “Открытие” виртуальной памяти внесло огромную контрибуцию в развитие современных технологий, облегчило работу как профессионального программиста, так и обычного пользователя, обеспечивая процесс более эффективного решения задач на ЭВМ[1].

К основным преимуществам виртуальной памяти относят:
1. избавление программиста от необходимости управлять общим пространством памяти,
2. повышение безопасность использования программ за счет выделения памяти,
3. возможность иметь в распоряжении больше памяти, чем это может быть физически доступно на компьютере.

Страничная организация памяти

При страничной организации все ресурсы памяти, как оперативной, так и внешней представляются для пользователя единым целым. Пользователь работает с общим адресным пространством и не задумывается какая память при этом используется: оперативная или внешняя, а эта общая память носит название виртуальной

(моделируемой). Виртуальная память разбивается на страницы, которые содержат определённое фиксированное количество ячеек памяти. При этом одна страница математической памяти не может быть больше или меньше других, все страницы должны быть одинаковы по количеству ячеек. Типичные размеры страниц 256, 512, 1024, 2048 Байт и более (числа кратные 256).



Сегментно-страничная организация виртуальной памяти

Данный метод организации виртуальной памяти направлен на сочетание достоинств **страничного и сегментного** методов управления памятью. В такой комбинированной системе адресное пространство пользователя разбивается на ряд сегментов по усмотрению программиста. Каждый сегмент в свою очередь разбивается на страницы фиксированного размера, равные странице физической памяти. С точки зрения программиста, логический адрес в этом случае состоит из номера сегмента и смещения в нем. Каждый сегмент представляет собой последовательность адресов от нуля до определённого максимального значения. Отличие сегмента от страницы состоит в том, что длинна сегмента может изменяться в процессе работы. Сегменты, как и любая структура виртуальной памяти, могут размещаться как в оперативной памяти, так и во внешней памяти (магнитных носителях). Виртуальная память с сегментно-страничной организацией функционирует подобно виртуальной памяти со страничной организацией: если требующийся на данный момент сегмент отсутствует в оперативной памяти, то при надобности работы с ним, он предварительно перемещается в ОП. Сегментно-страничная организация памяти требует более сложной аппаратурно-программной организации.

Таблицы страниц используются для перевода виртуальных адресов в физические адреса, используемые аппаратными средствами для обработки инструкций; такое аппаратное обеспечение, который

обрабатывает этот конкретный перевод часто называют блоком управления памятью. Каждая запись в таблице страниц держит флажок, указывающий, находится ли соответствующая страница в оперативной памяти или нет. Если она находится в оперативной памяти, запись в таблице страниц будет содержать реальный адрес памяти, где хранится страница[3]. Системы могут иметь как одну таблицу страниц для всей системы, так и отдельные таблицы страниц для каждого приложения и сегмента, деревья таблиц страниц для больших сегментов или некоторой их комбинации. Если есть только одна таблица страниц, различные приложения, работающие одновременно используют различные части одного диапазона виртуальных адресов. При наличии нескольких страниц или сегментов таблицы, есть несколько виртуальных адресных пространств и параллельных приложений с помощью отдельных таблиц страниц для перенаправления на другие реальные адреса.

MMU:

Блок управления памятью или **устройство управления памятью** (англ. *memory management unit*, MMU) — компонент аппаратного обеспечения компьютера, отвечающий за управление доступом к памяти, запрашиваемым центральным процессором.
Его функции заключаются в трансляции адресов виртуальной памяти в адреса физической памяти (то есть управление виртуальной памятью), защите памяти, управлении кэш-памятью, арбитражем шины и, в более простых компьютерных архитектурах (особенно 8-битных), переключением блоков памяти. Иногда также упоминается как блок управления страничной памятью (англ. *Paged memory management unit*, PMMU).
В настоящее время чаще всего упоминается в связи с организацией виртуальной памяти и, следовательно, критически важен для многих современных многозадачных операционных систем, включая все современные Windows NT и многие из UNIX-подобных. Специальная редакция ядра Linux, µClinux, может работать без MMU.
Блок управления памятью в настоящее время очень часто включается в состав центрального процессора или чипсета компьютера.

Принципы работы

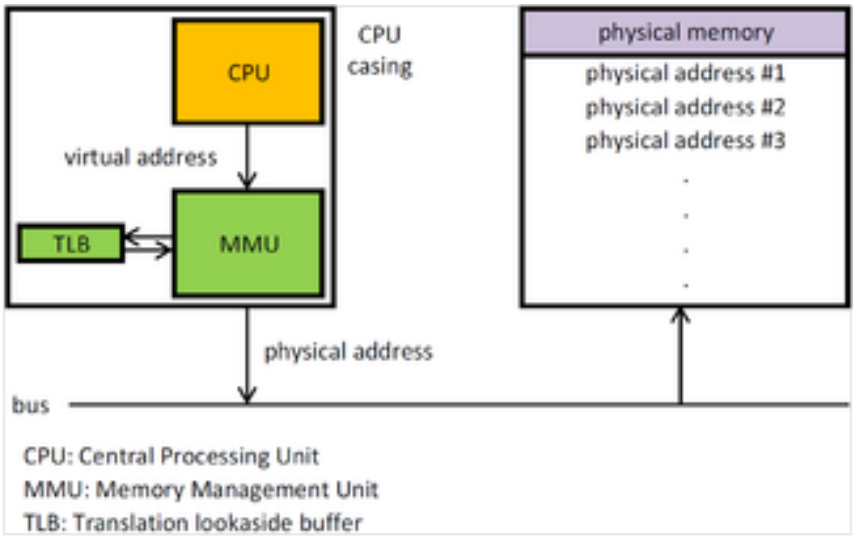


Схема работы MMU

Принцип работы современных MMU основан на разделении виртуального адресного пространства (одномерного массива адресов, используемых центральным процессором) на участки одинакового, как правило, несколько килобайт, хотя, возможно, и существенно большего, размера, равного степени 2, называемые страницами. Младшие n бит адреса (смещение внутри страницы) остаются неизменными. Старшие биты адреса представляют собой номер (виртуальной) страницы. MMU обычно преобразует номера виртуальных страниц в номера физических страниц, используя буфер ассоциативной трансляции (англ. Translation Lookaside Buffer, TLB).

Если преобразование при помощи TLB невозможно, включается более медленный механизм преобразования, основанный на специфическом аппаратном обеспечении или на программных системных структурах. Данные в этих структурах, как правило, называются элементами таблицы страниц (англ. page table entries (PTE)), а сами структуры — таблицами страниц (англ. page table (PT)). Конкатенация номера физической страницы со смещением внутри страницы даёт физический адрес.

Элементы PTE или TLB могут также содержать дополнительную информацию: бит признака записи в страницу (англ. dirty bit), время последнего доступа к странице (англ. accessed bit, для реализации алгоритма замещения страниц наиболее давно использованный (англ. least recently used, LRU), какие процессы (пользовательские (англ. user mode) или системные (англ. supervisor mode)) могут читать или записывать данные в страницу, необходимо ли кэшировать страницу.

TLB

Буфер ассоциативной трансляции (англ. *translation lookaside buffer, TLB*) — это специализированный кэш центрального процессора, используемый для ускорения трансляции адреса виртуальной памяти в адрес физической памяти.

TLB используется всеми современными процессорами с поддержкой страничной организации памяти. TLB содержит фиксированный набор записей (от 8 до 4096) и является ассоциативной памятью. Каждая запись содержит соответствие адреса страницы виртуальной памяти адресу физической памяти. Если адрес отсутствует в TLB, процессор обходит таблицы страниц и сохраняет полученный адрес в TLB, что занимает в 10 — 60 раз больше времени, чем получение адреса из записи, уже закэшированной TLB. Вероятность промаха TLB невысока и составляет в среднем от 0,01 % до 1 %.

TLB иногда реализуется как память с адресацией по содержимому (CAM). Ключ поиска CAM - это виртуальный адрес, а результат поиска - это физический адрес . Если запрошенный адрес присутствует в TLB, поиск CAM быстро дает совпадение, и полученный физический адрес может использоваться для доступа к памяти. Это называется попаданием TLB. Если запрошенный адрес отсутствует в TLB, это пропуск, и преобразование продолжается путем поиска таблицы страниц в процессе, называемом обходом *страниц*.. Обход страницы занимает много времени по сравнению со скоростью процессора, поскольку он включает чтение содержимого нескольких ячеек памяти и их использование для вычисления физического адреса. После того, как физический адрес определен обходом страниц, отображение виртуального адреса на физический адрес вводится в TLB. PowerPC 604 , например, имеет двухсторонний ассоциативно-набор TLB для загрузки данных и магазинов. [2] Некоторые процессоры имеют разные TLB адресов команд и данных.

3. **Системные утилиты сбора статистики ядра**

- Процессор: ps, top, tiptop, turbostat, rdmsr, numastat, uptime
  - ps: report a snapshot of the current processes

tty* это виртуальные консоли, ttyS — последовательные порты. /dev/pts/* это псевдотерминалы. В обычных терминалах есть ядро и есть процесс, работающий с этим терминалом. Процесс читает/

пишет данные в /dev/tty*, а ядро помещает туда ввод с клавиатуры и выводит данные на экран. В случае с псевдотерминалом всё, что читает/пишет процесс в /dev/pts/* обрабатывается другим процессом, например sshd или xterm. Ядро просто передаёт данные от одного процесса другому.

By default, **ps** selects all processes with the same effective user ID (euid=EUID) as the current user and associated with the same terminal as the invoker. It displays the process ID (pid=PID), the terminal associated with the process (tname=TTY), the cumulated CPU time in [DD-]hh:mm:ss format (time=TIME), and the executable name (ucmd=CMD). Output is unsorted by default.

- top: top - display Linux processes
-

- Виртуальная память: vmstat, slabtop, pidstat, free
- Дисковая подсистема: iostat, iotop, blktrace
- Сеть: netstat, tcpdump, iptraf, ethtool, nicstat, ip
- Интерактивные (типа top) или с указанием количества запуска и интервала (типа sar)
- Некоторые работают только с правами root!

Вопрос 1:
Зачем нужны страницы без прав в pmap (guard страницы),
Как просигнализируется ОС, если мы попытаемся туда записать, или прочесть? (Не системный вызов, не обращение к ядру напрямую)

This flag is used for stacks. It indicates to the kernel virtual memory system that the mapping should extend downward in memory. The return address is one page lower than the memory area that is actually created in the process's virtual address space. Touching an address in the "guard" page below the mapping will cause the mapping to grow by a page. This growth can be repeated until the mapping grows to within a page of the high end of the next lower mapping, at which point touching the "guard" page will result in a **SIGSEGV** signal.

If a thread overflows its
stack into the guard area, then, on most hard architectures, it receives a **SIGSEGV** signal, thus notifying it of the overflow. Guard areas start on page boundaries, and the guard size is internally rounded up to the system page size when creating a thread. (Nevertheless, **pthread_attr_getguardsize**() returns the guard size that was set by **pthread_attr_setguardsize**().)

These guard pages are unmapped pages placed between all memory allocations of one page or larger. The guard page causes a segmentation fault upon any access.

Эти защитные страницы представляют собой неотображенные страницы, размещенные между всеми выделениями памяти размером в одну страницу или больше. Страница защиты вызывает ошибку сегментации при любом доступе. В результате любая попытка злоумышленника перезаписать соседнюю память в ходе эксплуатации переполнения буфера приводит к завершению уязвимой программы, а не к продолжению выполнения кода, предоставленного злоумышленником. *Страницы защиты реализуются с помощью ряда систем и инструментов, включая OpenBSD, Electric Fence и Application Verifier (каждый из которых обсуждается далее в этой области содержимого).*
Страницы защиты имеют высокую степень накладных расходов, поскольку они фрагментируют карту памяти ядра и могут значительно увеличить объем виртуального пространства. Их эффективность зависит от размера и схемы распределения; они часто более эффективны как средство отладки, чем как средство оперативной безопасности.

Linux и другие системы на базе Unix используют механизм сигналов для уведомления применение в исключительных ситуациях. Можно назначить обработчик практически для всех типов сигналов. Доступ к запрещенному адресу будет перехвачен операционной системой, которая передаст приложению сигнал SIGSEGV. В этой ситуации довольно часто можно увидеть сообщение об ошибке Segmentation fault.

The OS allocates some space to the stack. When the process accesses an unallocated part of the stack, a page fault is raised by the processor and catched by the OS. If the OS believes it's still reasonable to grow the stack, it simply allocates new space for it and returns control to the process. If it's not reasonable, a stack overflow exception is raised.

1. For each running process, Linux keeps a list of regions of the virtual memory addresses. If an address reference generates a page fault, Linux checks that list to see if the virtual address is legal (within the range of one of the regions). If not claimed by a region, the application gets a SIGSEGV error, otherwise the kernel allocates another page of system memory and adds to the translation caches. if the faulting address just misses a region, and that region is for a stack (which grow up or down, according to the machine architecture) then Linux allocates another VM page, maps it into the region, thus growing the stack.
2. The kernel does not protect the stack. If a stack access causes a page fault because a physical VM page is not attached to a memory region for the process, the process's rlimit is tested to see if adding another page is permitted.
3. Stack guard pages are using by some malloc(3) debugger libraries. What these to is to expand each memory request by 2 VM pages: one page before the new page, one page after it. The extra pages are marked as no-access-at-all so if the application walks off the end of a region, or moves before the beginning, the application gets an access violation.

How is virtual address translated to physical address?

The CPU manages translation of virtual to physical addresses using **its Memory Management Unit (MMU)**. A virtual address is specified as a offset from the start of a memory segment; these segments are used by the kernel and user processes to hold their text, stack, data, and other regions.

The TLB is a cache that holds (likely) recently used pages. The principle of locality says that the pages referenced in the TLB are likely to be used again soon. This is the underlying idea for all caching. When these pages are needed again, it takes minimal time to find the address of the page in the TLB. The page table itself can be enormous, so walking it to find the address of the needed page can get very expensive.

the page tables are stored in **the kernel address space**. Each process has its own page table structure, which is set up so that the kernel portion of the address space is shared between processes. The kernel address space is not accessible from user space, however.

План:

1. Зачем нужны
2. эти защитные страницы представляют собой неотображенные страницы, размещенные между всеми выделениями памяти размером в одну страницу или больше. Страница защиты вызывает ошибку сегментации при любом доступе. В результате любая попытка злоумышленника перезаписать соседнюю память в ходе эксплуатации переполнения буфера приводит к завершению уязвимой программы, а не к продолжению выполнения кода, предоставленного злоумышленником
3. Linux и другие системы на базе Unix используют механизм сигналов для уведомления применение в исключительных ситуациях. Можно назначить обработчик практически для всех типов сигналов. Доступ к запрещенному адресу будет перехвачен операционной системой, которая передаст приложению сигнал SIGSEGV. В этой ситуации довольно часто можно увидеть сообщение об ошибке Segmentation fault.
4. causes a page fault because a physical VM page is not attached to a memory region for the process: mmu, tlb, page table

- MMU обычно преобразует номера виртуальных страниц в номера физических страниц, используя буфер ассоциативной трансляции (англ. Translation Lookaside Buffer, TLB).
- Если преобразование при помощи TLB невозможно, включается более медленный механизм преобразования, основанный на специфическом аппаратном обеспечении или на программных системных структурах. Данные в этих структурах, как правило, называются элементами таблицы страниц (англ. page table entries (PTE)), а сами структуры — таблицами страниц (англ. page table (PT)). Конкатенация номера физической страницы со смещением внутри страницы даёт физический адрес.

Вопрос 2:
Есть ld.so библиотеки. Ld библиотеки это круто, мы экономим место, если она где-то

используется ещё, она не будет подгружаться второй раз. Но она лежит в адресном пространстве другого процесса, как другой процесс получает к ней доступ , если адресные пространства изолированы.

A dynamic linker, in contrast, inserts symbolic references to (library) modules in the executable file, and leaves these to be resolved at run time. In operating systems conforming to the Unix System-V interface [X/O90], such as SGI's Irix, DEC's Digital Unix or Sun's Solaris-2, this works as follows.

For each library module to be linked, the linker allocates a global offset table (GOT).1 The GOT contains the addresses of all dynamically linked external symbols (functions and variables) referenced by the module.2 When a program referencing such a dynamically linked module is loaded into memory for execution, the imported module is loaded (unless already resident) and a region in the new process' address space is allocated in which to map the module. The loader then initialises the module's GOT (which may require first loading other library modules referenced by the module just loaded).

A variant of this is lazy loading, where library modules are not actually loaded until accessed by the process. If lazy loading is used, a module's GOT is initialised at module load time with pointers to stub code. These stubs, when called, invoke the dynamic loader, which loads the referenced module and then replaces the respective entries in the GOT by the addresses of the actual variables and functions within the newly loaded module. Dynamic linking has a number of advantages over static

linking: 1. Library code is not duplicated in every executable image referencing it. This saves significant amounts of disk space (by reducing the size of executable files) and physical memory (by sharing library code between all invocations). These savings can be very substantial and have significant impact on the performance of the virtual memory system. 2. New (and presumably improved) versions of libraries can be installed and are immediately usable by client programs without requiring explicit relinking. 3. Library code which is already resident can be linked immediately, thus reducing process startup latency. Lazy loading further reduces startup cost, at the expense of briefly stalling execution when a previously unaccessed library module requires loading. For libraries which

are only occasionally used by the program, this results in an overall speedup for runs which do not access the library.

However, this comes at a cost: Should the referenced library be unavailable (e.g., by having been removed since program link time) this may only become evident well into the execution of the program. Many users will prefer finding out about such error conditions at program startup time. The main drawbacks of dynamic linking are: 1. Extra work needs to be done at process instantiation to set up the GOT. However, this overhead is easily amortised by loading much less code in average, as some libraries will already be resident. 2. If a dynamic library is (re)moved between link and load time, execution will fail. This is the main reason that Unix systems keep static linking as an option. 3. The location of a dynamically linked module in the process' address space is not known until run time, and the same module will, in general, reside at different locations in different clients' address spaces. This requires that dynamically linked libraries only contain position-independent code. 3 Position independence requires that all jumps must
be PC-relative, relative to an index-register containing the base address of the module, or indirect (via the GOT).

The main cost associated with positionindependent code is that of locating the GOT. Every exported ("public") function in the module must first locate the module's GOT. The GOT is allocated at a constant offset from the function's entry point (determined by the linker) so the function can access it using PC-relative addressing. This code must be executed at the beginning of every exported function. In addition, there is an overhead (of one cycle) for calling the function, as an indirect jump must be used, rather than jumping to an absolute address. These costs will be examined further in Section 6.

Напротив, динамический компоновщик вставляет символические ссылки на (библиотечные) модули в исполняемый файл и оставляет их для разрешения во время выполнения. В операционных системах, соответствующих интерфейсу Unix System-V [X / O90], таких как SGI Irix, DEC Digital Unix или Sun Solaris-2, это работает следующим образом.

Для каждого библиотечного модуля, который должен быть связан, компоновщик выделяет глобальную таблицу смещений (GOT) .1 GOT содержит адреса всех динамически связанных внешних символов

(функций и переменных), на которые ссылается модуль. Модуль загружается в память для выполнения, загружается импортированный модуль (если он еще не установлен), и выделяется область в адресном пространстве нового процесса, в которую следует сопоставить модуль. Затем загрузчик инициализирует GOT модуля (для чего может потребоваться сначала загрузить другие библиотечные модули, на которые ссылается только что загруженный модуль).

The **Global Offset Table**, or **GOT**, is a section of a computer program's (executables and shared libraries) memory used to enable computer program code compiled as an ELF file to run correctly, independent of the memory address where the program's code or data is loaded at runtime.[1] It maps symbols in programming code to their corresponding absolute memory addresses to facilitate Position Independent Code (PIC) and Position Independent Executables (PIE)[2] which are loaded[3] to a different memory address each time the program is started. The runtime memory address, also known as absolute memory address of variables and functions is unknown before the program is started when PIC or PIE code is run[4] so cannot be hardcoded during compilation by a compiler. The Global Offset Table is represented as the .got and .got.plt sections in an ELF file[5] which are loaded into the program's memory at startup.[5] [6] The operating system's dynamic linker updates the global offset table relocations (symbol to absolute memory addresses) at program startup or as symbols are accessed.[7] It is the mechanism that allows shared libraries (.so) to be relocated to a different memory address at startup and avoid memory address conflicts with the main program or other shared libraries, and to harden computer program code from exploitation.[8]

So what is the Global offset Table (GOT)? The Global Offset Table redirects position independent address calculations to an absolute location and is located in **the . got section of an ELF executable or shared object**. It stores the final (absolute) location of a function calls symbol, used in dynamically linked code.

Вариантом этого является отложенная загрузка, при которой модули библиотеки фактически не загружаются до тех пор, пока к ним не обращается процесс. Если используется отложенная загрузка, GOT модуля инициализируется во время загрузки модуля указателями на код-заглушку. Эти заглушки при вызове вызывают динамический загрузчик, который загружает указанный модуль, а затем заменяет

соответствующие записи в GOT адресами фактических переменных и функций во вновь загруженном модуле. Динамическое связывание имеет ряд преимуществ перед статическим

связывание: 1. Код библиотеки не дублируется в каждом исполняемом образе, ссылающемся на него. Это экономит значительное количество дискового пространства (за счет уменьшения размера исполняемых файлов) и физической памяти (за счет совместного использования кода библиотеки между всеми вызовами). Эта экономия может быть очень значительной и существенно повлиять на производительность системы виртуальной памяти. 2. Могут быть установлены новые (и предположительно улучшенные) версии библиотек, которые могут быть немедленно использованы клиентскими программами без необходимости явного повторного связывания. 3. Библиотечный код, который уже находится в памяти, можно связать немедленно, тем самым уменьшая задержку запуска процесса. Ленивая загрузка дополнительно снижает затраты на запуск за счет кратковременной остановки выполнения, когда требуется загрузка ранее недоступного библиотечного модуля. Для библиотек, которые используются программой лишь изредка, это приводит к общему ускорению запусков, которые не обращаются к библиотеке.

Однако за это приходится платить: если указанная библиотека недоступна (например, из-за того, что она была удалена после времени компоновки программы), это может стать очевидным только при выполнении программы. Многие пользователи предпочтут узнавать о таких состояниях ошибки во время запуска программы. Основными недостатками динамического связывания являются: 1. При создании экземпляра процесса необходимо выполнить дополнительную работу для настройки GOT. Однако эти накладные расходы легко окупаются за счет загрузки в среднем гораздо меньшего количества кода, поскольку некоторые библиотеки уже будут резидентными. 2. Если динамическая библиотека перемещается (повторно) между временем компоновки и загрузкой, выполнение завершится ошибкой. Это основная причина того, что системы Unix поддерживают статическое связывание в качестве опции. 3. Местоположение динамически связанного модуля в адресном пространстве процесса неизвестно до времени выполнения, и один и тот же модуль, как правило, будет находиться в разных местах в адресных пространствах разных клиентов. Для этого необходимо, чтобы динамически подключаемые библиотеки содержали только позиционно-независимый код. 3 Независимость позиции требует, чтобы все переходы были относительны к ПК,

относительно индексного регистра, содержащего базовый адрес модуля, или косвенно (через
GOT).
Основные затраты, связанные с позиционно-независимым кодом, связаны с определением местоположения GOT.
Каждая экспортированная («общедоступная») функция в модуле сначала необходимо найти GOT модуля. GOT - это размещается с постоянным смещением от функции точка входа (определяется компоновщиком), поэтому функция может получить к нему доступ, используя адресацию относительно ПК.
Этот код должен выполняться в начале каждая экспортируемая функция. Кроме того, есть накладные расходы (одного цикла) на вызов функции, как непрямой прыжок должен использоваться, а не переход к абсолютному адресу. Эти расходы будут более подробно рассматривается в Разделе 6.

Сегмент кода разделяемой библиотеки существует в памяти в единственном экземпляре для *каждой системы* . Тем не менее, он может быть сопоставлен с разными виртуальными адресами для разных процессов, поэтому разные процессы видят одну и ту же функцию по разным адресам (поэтому код, который идет в общую библиотеку, должен быть скомпилирован как PIC).
Сегмент данных общей библиотеки создается в одной копии для каждого процесса и инициализируется любыми начальными значениями, указанными в библиотеке.
Это означает, что вызывающим объектам библиотеки не нужно знать, является ли она общей или нет: все вызывающие объекты в одном процессе видят одну и ту же копию функций и одну и ту же копию внешних переменных, определенных в библиотеке.
Разные процессы выполняют один и тот же код, но имеют свои индивидуальные копии данных, по одной копии на процесс.

Динамические и разделяемые библиотеки обычно одинаковы. Но в вашем случае это выглядит так, будто вы делаете что-то особенное.

- В случае с *общей библиотекой* вы указываете общую библиотеку во время компиляции. Когда приложение запускается, операционная система загружает общую библиотеку перед запуском приложения.

- В случае *динамической библиотеки* библиотека не указывается во время компиляции, поэтому она не загружается операционной системой. Вместо этого ваше приложение будет содержать некоторый код для загрузки библиотеки.

Первый случай - нормальный случай. Второй случай - это особый случай, и он в основном актуален, если ваше приложение поддерживает такие расширения, как **плагины** . Динамическая загрузка необходима, потому что плагинов может быть много, и они создаются после вашего приложения. Таким образом, их имена недоступны во время компиляции.

- static linking: the build-time linker (ld) resolves all the objects used in the program during the build, merges the objects which are used, and produces an executable binary which doesn't use external libraries;
- dynamic linking, at build-time: ld resolves all objects used in the program, but instead of storing them in the executable, it only stores references to them;
- dynamic linking, at run-time: the run-time linker (ld.so), or dynamic linker, resolves all the references stored in the executable, loading all the required libraries and updating all the object references before running the program.

По сути, каждая разделяемая библиотека, содержащая статические данные (например, глобальные переменные), имеет глобальную таблицу смещения (GOT) . В разделяемых библиотеках все ссылки на статические данные (подумайте о глобальных варах) происходят через GOT (они косвенные). Таким образом, даже если сегмент кода совместно используется несколькими процессами, каждый процесс имеет свое эксклюзивное отображение других сегментов общей библиотеки, включая соответствующую GOT, записи которой перемещаются соответствующим образом.
Короче говоря, **между процессами используется только код, а не данные** . Однако я думаю, что константы могут быть исключением в зависимости от флагов компиляции.

Чтобы подчеркнуть этот момент, unixy-система может либо

совместно использовать, либо не совместно использовать динамическую библиотеку, но с точки зрения приложения *нет заметной разницы* между обеими реализациями. Почти все unix-подобные системы используют общий код между процессами, потому что это легко сделать и это отличный способ сэкономить оперативную память практически бесплатно. Редкие исключения - параноидальные операционные системы на оборудовании со слабым (или отсутствующим) MMU, так что общий доступ textможет позволить одному процессу повредить другой.

Код общей библиотеки копируется (или, точнее, отображается) в память операционной системой. Затем ОС предоставляет каждому из процессов доступ к этой копии в памяти.
Возможно, что каждый из процессов «увидит» копию как находящуюся по другому адресу памяти, чем другой. Это решается блоком управления памятью ЦП.
Это может быть более сложным, чем это, но в основном так все работает в Linux и других связанных с Unix операционных системах, таких как Mac OS X.

Эти два процесса используют один и тот же физический адрес сегмента кода совместно используемой библиотеки, но их виртуальный адрес отличается для двух процессов. Виртуальная память помогает реализовать эту функцию здесь.

Согласно книге « Компьютерные системы: взгляд программиста» в главе 9.5. *ВМ как инструмент для управления памятью.*
Однако в некоторых случаях желательно, чтобы процессы совместно использовали код и данные. Например, каждый процесс должен вызывать один и тот же код ядра операционной системы, а каждая программа на C вызывает подпрограммы стандартной библиотеки C, такие как printf. Вместо того, чтобы включать отдельные копии ядра и стандартной библиотеки C в каждый процесс, операционная система может организовать совместное использование одной копией этого кода несколькими процессами путем сопоставления соответствующих виртуальных страниц в разных процессах с одними и теми же физическими страницами.

План:

1. Отличие статической библиотеки от динамической:
    - static linking: the build-time linker (ld) resolves all the objects used in the program during the build, merges the objects which are used, and produces an executable binary which doesn’t use external libraries;
    - dynamic linking, at build-time: ld resolves all objects used in the program, but instead of storing them in the executable, it only stores references to them;
    - dynamic linking, at run-time: the run-time linker (ld.so), or dynamic linker, resolves all the references stored in the executable, loading all the required libraries and updating all the object references before running the program.
2. Преимущество динамических библиотек:
    - 1. Код библиотеки не дублируется в каждом исполняемом образе, ссылающемся на него. Это экономит значительное количество дискового пространства (за счет уменьшения размера исполняемых файлов) и физической памяти (за счет совместного использования кода библиотеки между всеми вызовами). Эта экономия может быть очень значительной и существенно повлиять на производительность системы виртуальной памяти.
    - 2. Могут быть установлены новые (и предположительно улучшенные) версии библиотек, которые могут быть немедленно использованы клиентскими программами без необходимости явного повторного связывания.
    - 3. Библиотечный код, который уже находится в памяти, можно связать немедленно, тем самым уменьшая задержку запуска процесса. Ленивая загрузка дополнительно снижает затраты на запуск за счет кратковременной остановки выполнения, когда требуется загрузка ранее недоступного библиотечного модуля. Для библиотек, которые используются программой лишь изредка, это приводит к общему ускорению запусков, которые не обращаются к библиотеке. Однако за это приходится платить: если указанная библиотека недоступна (например, из-за того, что она была удалена после времени компоновки программы), это может стать очевидным только при выполнении программы.
3. Недостатки:
    - 1. При создании экземпляра процесса необходимо выполнить дополнительную работу для настройки GOT. Однако эти накладные расходы легко окупаются за счет загрузки в

среднем гораздо меньшего количества кода, поскольку некоторые библиотеки уже будут резидентными.
    - ~~2. Если динамическая библиотека перемещается (повторно) между временем компоновки и загрузкой, выполнение завершится ошибкой. Это основная причина того, что системы Unix поддерживают статическое связывание в качестве опции.~~
    - 3. Местоположение динамически связанного модуля в адресном пространстве процесса неизвестно до времени выполнения, и один и тот же модуль, как правило, будет находиться в разных местах в адресных пространствах разных клиентов. Для этого необходимо, чтобы динамически подключаемые библиотеки содержали только позиционно-независимый код.
    - 3 Независимость позиции требует, чтобы все переходы были относительны к ПК, относительно индексного регистра, содержащего базовый адрес модуля, или косвенно (через GOT).
4. GOT и где размещаются исходники библиотеки:
    - Напротив, динамический компоновщик вставляет символические ссылки на (библиотечные) модули в исполняемый файл и оставляет их для разрешения во время выполнения.
    - Для каждого библиотечного модуля, который должен быть связан, компоновщик выделяет глобальную таблицу смещений (GOT) .1 GOT содержит адреса всех динамически связанных внешних символов (функций и переменных), на которые ссылается модуль. Модуль загружается в память для выполнения, загружается импортированный модуль (если он еще не установлен), и выделяется область в адресном пространстве нового процесса, в которую следует сопоставить модуль. Затем загрузчик инициализирует GOT модуля
    - **Global Offset Таблица** или **GOT** , является сечение компьютерной программы «s (исполняемые файлы и разделяемые библиотеки) памяти , используемой для того, чтобы компьютерный программный код , скомпилированный как ELF - файла для запуска правильно, не зависит от адреса памяти , где код программы или данные будет загружен во время выполнения. [1]Он сопоставляет символы в программном коде с соответствующими абсолютными адресами памяти, чтобы облегчить позиционно-независимый код (PIC) и позиционно-независимые исполняемые файлы (PIE) [2], которые

загружаются [3] в разные адреса памяти при каждом запуске программы. Адрес памяти времени выполнения, также известный как абсолютный адрес памяти переменных и функций, неизвестен до запуска программы при запуске кода PIC или PIE [4], поэтому не может быть жестко закодирован во время компиляции компилятором . Глобальная таблица смещения представлена в виде разделов .got и .got.plt в файле ELF [5], которые загружаются в память программы при запуске. [5] [6] Динамический компоновщик операционной системы обновляет перемещения таблицы глобальных смещений (символы в абсолютные адреса памяти) при запуске программы или при доступе к символам. [7] Это механизм, который позволяет разделяемым библиотекам (.so) перемещаться по другому адресу памяти при запуске и избегать конфликтов адресов памяти с основной программой или другими разделяемыми библиотеками, а также защищать код компьютерной программы от эксплуатации. [8]
    - По сути, каждая разделяемая библиотека, содержащая статические данные (например, глобальные переменные), имеет глобальную таблицу смещения (GOT) . В разделяемых библиотеках все ссылки на статические данные (подумайте о глобальных варах) происходят через GOT (они косвенные). Таким образом, даже если сегмент кода совместно используется несколькими процессами, каждый процесс имеет свое эксклюзивное отображение других сегментов общей библиотеки, включая соответствующую GOT, записи которой перемещаются соответствующим образом.
5. Как другие процессы могут получить доступ к библиотеке

- Сегмент кода разделяемой библиотеки существует в памяти в единственном экземпляре для *каждой системы* . Тем не менее, он может быть сопоставлен с разными виртуальными адресами для разных процессов, поэтому разные процессы видят одну и ту же функцию по разным адресам (поэтому код, который идет в общую библиотеку, должен быть скомпилирован как PIC).
- Сегмент данных общей библиотеки создается в одной копии для каждого процесса и инициализируется любыми начальными значениями, указанными в библиотеке.
- Это означает, что вызывающим объектам библиотеки не нужно знать, является ли она общей или нет: все вызывающие объекты в одном процессе видят одну и ту же копию функций и одну и ту же копию внешних переменных, определенных в библиотеке.
- Разные процессы выполняют один и тот же код, но имеют свои индивидуальные копии данных, по одной копии на процесс.

6. Ключевое:

**Код общей библиотеки копируется (или, точнее, отображается) в память операционной системой.
Затем ОС предоставляет каждому из процессов доступ к этой копии в памяти.
Возможно, что каждый из процессов «увидит» копию как находящуюся по другому адресу памяти, чем другой. Это решается блоком управления памятью ЦП.
Это может быть более сложным, чем это, но в основном так все работает в Linux и других связанных с Unix операционных системах, таких как Mac OS X.**

Эти два процесса используют один и тот же физический адрес сегмента кода совместно используемой библиотеки, но их виртуальный адрес отличается для двух процессов. Виртуальная память помогает реализовать эту функцию здесь.