29.10.2024г.

Тема 11: Управление на процеси: fork(2), exit(3), atexit(3), wait(2), exec(3), getppid(2), getpid(2).

1) fork() - създава точно копие на процеса, който прави извикването на функцията. Двете копия започват да работят паралелно, като във всяко от тях се изпълнява следващата по ред инструкция.

! Двете копия продължават да изпълняват програмата оттам, докъдето са стигнали. Единствената разлика в двете копия е във върнатата стойност на fork()

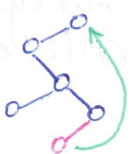
pid_t child = fork()

- родителят получава в променливата child номера на породения процес (детето).
- при детето резултатът е 0
- ако fork() не може да се изпълни, резултатът от извикването на командата е -1

? Кога няма да може да се изпълни fork():
Когато някакъв системен ресурс е изчерпан - това може да е:
- паметта - т.е. няма памет за копието на процеса
- таблицата на процесите / кореновото дърво да е препълнено
- може и да е заради ограничения на самата ОС: съвременните UNIX-и имат системи за ограничаване ресурсите на потребители и групи / напр. на един потребител може да е разрешено да стартира само 20 процеса /

Системното извикване fork() е просто - то няма параметри.
Интересната му употреба е с други системни извиквания: напр. ако искаме да стартираме процес при определени условия - дете и родител да работят по различен начин. Така че fork() обикновено се използва в комбинация с някое друго системно извикване.

! В системата UNIX, в стандарта POSIX се поддържа информация за това кой процес на кого е родител. Това е съвсем стандартно, обикновено кореново дърво.



Когато fork() сработи за някакъв процес, се създава дете и в това кореново дърво се появява ново ребро.
Тази информация се пази, докато двата процеса са „живи"

1) Какво става, когато родителят прекъсне своята работа?
Ако поради някакви причини родителят прекъсне своята работа, а детето още работи, тогава при изчезване на родителя детето получава нов родител, който е процесът с номер 1 - първият процес в кореновото дърво (init).

2) Какво става, когато детето прекъсне своята работа?
Детето вече не съществува като процес, но реброто остава, кореновото дърво се запазва същото. От детето към родителя се изпраща сигналът SIGCHLD. Родителят следва да обработи сигнала със специалното системно извикване wait() и да получи кода на завършване на процеса дете. ! Това е !-ният начин кодът на завършване / кодът за грешка да бъде получен от родителя.

Докато не се извърши тази обработка, процесът дете, което е приключило, остава в състояние zombie - повече не заема други системни ресурси освен реброто. Ако в дървото останат много зомбита, то ще се препълни - затова трябва да се трият редовно.

„Древната" и „модерната" реализация на fork

* Класическото извикване fork() просто създава копие на паметта на родителския процес. Всеки процес си има изолирана /локална/ памет - част от RAM-а, и на тази памет се прави точно копие.
Освен че детето получава точно копие на тази изолирана /локална/ памет, получава и същите fd-и, т.е. fd-ите са споделени обекти.
В класическата си реализация fork() е (в някакъв смисъл) бавен - цялата локална памет (стек, heap, fd-и,...) трябва да се копира. Данните може да са обемни и копирането да е бавно. По време на копирането процесът няколко сек.

* Съвременната реализация на fork() борави с VM (виртуалната памет).
Всеки процес притежава таблица, в която е описано кои страници от реалната памет използва. Съвременният fork() прави копие не на паметта, а на въпросната таблица (на мета данните, които описват VM) и я модифицират.

В резултат на fork() родителят и детето се намират на различни части от паметта, които обаче имат еднакво съдържание. ! Писане в паметта, и тар-ване или интар-ване на единия процес не се отразяват върху другия процес. ! На детето наследява fd-ите на родителя.

2) exit() - прекратява работата на процес (терминира го) по нормален начин; затваря fd-и

```
void exit (int Status);
```

Аргументът Status може да се зададе от потребителя. Ако никъде в програмата си не извикваме exit(), компилаторът слага exit() с нормален статус в края на кода:
→ при успешно завършване на програмата: exit(0)
→ при неуспешно завършване на програмата: exit(1).

Освен това има възможност по време на exit да се изпълнят и други функции, дефинирани от потребителя, наред със стандартните функции, които системата ще извърши (като затварянето на fd-ите и други).
Добавянето на такива допълнителни функции, които да се извършат при нормалното терминиране на процес (с exit()), става с atexit().

3) atexit()

```
int atexit( void (*function) (void));
```

Едно извикване на atexit се добавя една функция, която да се извика при exit() (създава се списък с тези ф-ции

Като първа ще се извиква последната регистрирана функция, т.е. извикват се в обратен ред на добавянето.

! Възможно е една функция да бъде регистрирана неколкократно → ще се извиква по веднъж за всяка регистрация.

Важно: При приключването на процеса (по нормален начин, с exit()) най-напред ще се изпълнят добавените с atexit() функции от потребителя и чак след тях стандартните функции за затваряне на програмата. С atexit() можем да добавим функция за дописване във файловете и други и по този начин да гарантираме, че няма да има загуба на данни.

4) wait() - чака някое дете да си смени състоянието, статуса - най-вече да приключи работа

```
pid_t wait(int *wstatus);
```

waitpid() - чака конкретно дете (с определено pid) да приключи

```
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

Основната функционалност на wait() е да получи кода на грешка на приключилия процес. Иначе след терминирането на процеса този код на грешка „виси", никъде не е регистриран, намира се някъде в корковото дърво. Целта е родителят да получи този код и да го погълне.
Детето изпраща сигнала SIGCHLD, а извикването wait() приема сигнала и кода на грешка
↳ записва се в променливата wstatus - аргумент на wait().
Върнатият от wait() резултат е pid на детето.

! wait() е приспиваща команда, тя блокира процеса родител.
В този режим на работа родителят бива приспан, не работи, докато детето не завърши.
принудително блокиран
За да работят паралелно двата процеса, процедурата, която обработва сигнала SIGCHLD (неговият signal handler) при приемането му, може да извърши wait(). Нормалната реакция на SIGCHLD е Ign. Трябва да се промени, да се дефинира нова функция, в която се извиква wait(). Така и двата процеса ще работят паралелно, а когато детето приключи сигналът SIGCHLD ще стартира новата, специалната процедура - тя за кратко време ще свърши своята работа, ще приеме кода на приключилия процес и след това ще възстанови нормалната работа на родителя.

Самият wstatus е голямо число, разделено е на „парчета", едното от които е кодът на завършване. Другите битове показват какви са били причините за завършването, и те могат да се изследват.

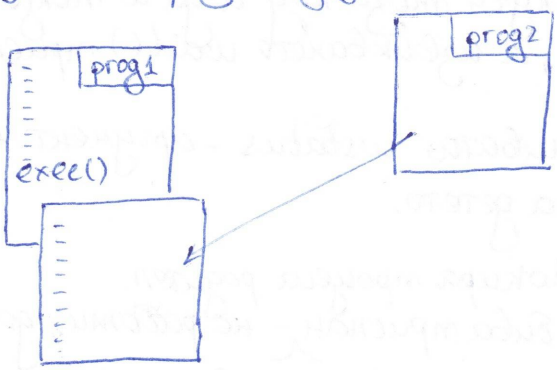
Ако wstatus не е NULL, wait() и waitpid() запазват статус информацията в променливата от тип int, към която wstatus сочи. Въпросният integer може да се изследва със следните макроси (като аргумент приемат самия integer, а не указателя му, както правят wait() и waitpid()):

1) WIFEXITED (wstatus) - макрос за нормално завършил наследник
2) WEXITSTATUS (wstatus) - макрос за статус на нормално завършил наследник
3) WIFSIGNALED (wstatus) - макрос за наследник, прекъснат от сигнал
4) WTERMSIG (wstatus) - макрос за сигнал, прекъснал наследника

1) ще върне „истина", ако детето е приключило нормално, т.е. ако сам си е извикало exit() или main() функцията му е завършила (което означава, че накрая се е извикал exit()).
2) Ако наследникът е завършил нормално (т.е. WIFEXITED връща true), този макрос връща статуса на детето.
3) връща „истина", ако процесът наследник е бил прекъснат от сигнал (! Някои сигнали предизвикват терминиране на процеса - това не е нормално завършване)
4) Ако наследникът е бил прекъснат от сигнал (т.е. WIFSIGNALED връща true), този макрос връща номера на сигнала.

---

5) exec() - заменя образа с друг



Тема 14: Изпращане и обработка на сигнали signal(7): signal(2), kill(2), pause(2), alarm(2).

Сигналите са абстракции (не са особено синхронизирани с останалите абстракции - файлове, пространство с атрибути и имена и процеси, конун. канали).
по-постоянни, статични обекти — динамични

Програмата може да реагира на аварийни събития /аварийни ситуации/ - това са асинхронни реакции! Докато се изпълнява кодът на програмата, някое аварийно събитие може да прекъсне изпълнението му и да предизвика аварийно действие. Абстракцията, която ни го осигурява, са UNIX сигналите. - тях може да разглеждаме като аналози на:

- хардуерните прекъсвания

Позволяват в една сложна компютърна конфигурация, в която има паралелно работещи устройства (цялото множество от периферни устройства и техни контролери и централния процесор, който изпълнява програмата / мн-во от централни процесори и ядра, които изпълняват паралелни програми), да има взаимодействие м/у тях, така че кем да работят паралелно, кем да си сигнализират, изпращат информация при настъпване на важни моменти във времето, за да синхронизират своята работа.

! Целта е паралелно работещите части на компютъра да могат да поддържат комуникация помежду си, за да могат успешно да си взаимодействат и правилно да подреждат операциите си във времето. ! Да е ефективна паралелната им работа.

Напр.: Програмата е поръчала на някое периферно у-во да свърши определена задача.
↳ поръчала е на диска да позиционира главата върху даден цилиндър и да прочете даден сектор. Тъкмо това е бавна операция, програмата продължава да се изпълнява. Чак когато дискът /неговият контролер/ изпълни зададената от програмата команда, вдига сигнал, хардуерно прекъсване. Прекъсва работата на програмата. Дискът съобщава, че е приключил с тази задача и може да подаде на програмата прочетения сектор, и е готов да изпълни следваща команда.

Друг пример: Клавиатурата подава електрически сигнал (при натиснат клавиш), предизвиква прекъсване, т.е. спира работата на програмата. Съобщава за наличие на натиснат клавиш, предоставя го за четене от програмата и така команда за четене на нов клавиш. ! Програмата не зацикля в очакване на натиснат клавиш. Процесите се изпълняват паралелно.

- софтуерните прекъсвания - аналог, но в средата, в която работят абстракциите на ОС. Създадени за поддържане независимост от хардуера. Стандартът POSIX осигурява една напълно виртуална среда, в която програмите, които програмистът пише, да не зависят от дребните хардуерни конфигурации и да съществуват „много години" в абстрактната ОС.

! Целта на софтуерните прекъсвания, основната им функция се крие в това един обект да може да прати аварийно съобщение до друг обект, че нещо специално се е случило, и другият обект да реагира.

Историята на прекъсванията в UNIX

1) В началото имало различни видове аварийни събития, като реакцията на всяка била специфична.

2) Няколко години по-късно възниква идеята да се реализират като концепция, такава че да приличат на хардуерните прекъсвания и да имат единообразна интерпретация.

В съвременния UNIX (5г. след създаването му) се формира концепцията

Сигналът е 1В, който даден активен обект (процес, периферно у-во) в ОС подава към един или група процеси. Когато този 1В достигне до съответния процес и ако има права да предизвиква реакция, я предизвиква. Спира се изпълнението на процеса и се задейства специфична реакция (в зависимост от вида на сигнала - не всички предизвикват реакция; В стандартния UNIX са около 20, в съвременните са 30-40).

! Важно: В момента на получаването на сигнала програмата спира своята работа и стартира специалната функция с име signal handler, която е реакция на прекъсването.

Ако програмата не бъде спряна аварийно или нормално, продължава нейното изпълнение от там, където е настъпило прекъсването.

* В програмните езици също понятието exception - същото е !!! Събитие, което прекъсва изпълнението на програмата и изпълнява функция, която ние сме дефинирали като реакция на exception-а.

Основни сигнали (архаични), които ние ползваме. Има и модерни, които са налични или не са налични в зависимост от конкретната ОС.

I. Реакции: има няколко стандартни реакции.

Ако не предефинираме реакция, се изпълнява стандартната за сигнала реакция

- Ign - не предизвиква никакво действие, програмата игнорира сигнала и продължава
- Term - процесът спира по нормален начин, все едно се изпълнява системната функция exit(). Програмата спира, докъдето е стигнала.
- Core - предизвиква аварийно спиране - при „неприятна" грешка
- Stop, Cont } специални реакции, които спират / събуждат процеса. По този начин външен източник може да нареди даден процес да бъде прекъснат. После може да се събуди с друлния сигнал.

Това са реакциите по подразбиране. Програмистът може да подмени тези със свои чрез ф-цията signal(). Изпращането на сигнал става с kill() ф-ция (3 параметъра)

С функцията signal() се подменя signalhandler.
Подяна на реакцията по подразбиране на даден сигнал с друга може да се из
върши и с функцията sigaction(), но това не е обект на изучаване на лекциите.

Необходимостта от сигнали (за кое) се свежда до:

1) възможността изобщо да се реагира на специални събития
2) контролиране на множество работещи процеси

Другите (основните) абстракции (файлове, пр-во на имената, процеси, канали - изградени в началото) не осигуряват механизъм за контрол на множество от процеси така стриктно.

* kill() - един процес изпраща сигнал до:
  - → друг процес;
  - → група процеси;
  - → всички процеси;

Забележка: Ако един процес има много опашки, сигналът може да бъде изпратен и само до една от опашките му, определена.

* Възможно е сигналите да се маскират; има сигнални маски
Може временно да се забрани получаването на определени сигнали - те ще бъдат запомнени, но няма да предизвикат реакция

? Как се изпълнява signalhandler-ът?

Сигналът се взима от опашката на чакащи сигнали, които са постъпили към процес и се изпълнява signalhandler. Обработката на сигнала може да се извършва в специален стек. След това се възстановява изпълнението на програмата (ако не е терминирана).

Стандартни сигнали

Всяка версия на UNIX има множество от стандартни сигнали. Нарича ме ги „стандартни", защото са описани в стандарта POSIX (има 2 други версии на стандарта POSIX: от 1990г. и от 2001г.).

сигнал:	ст-ст:	реакция:	пояснение:
SIGHUP	1	Term	прекъсване на връзката с управляващия терми
SIGINT	2	Term	„прекъсване" от клавиатурата (Ctrl+C)
SIGQUIT	3	Core	„излизане" от клавиатурата (Ctrl+\)
SIGILL	4	Core	недопустима инструкция
SIGABRT	6	Core	сигнал от abort()
SIGFPE	8	Core	прекъсване при операция с плаваща точка
SIGKILL	9	Term	сигнал за убиване
SIGSEGV	11	Core	недопустима операция с паметта
SIGPIPE	13	Term	писане в канал без читатели

сигнал	ст-ст	реакция	значение
SIGALRM	14	Term	сигнал от alarm()
SIGTERM	15	Term	сигнал за прекратяване
SIGUSR1	10	Term	потребителски сигнал 1
SIGUSR2	12	Term	потребителски сигнал 2
SIGCHLD	17	Ign	завършване на наследник
SIGCONT	18	Cont	продължава, ако е спрял
SIGSTOP	19	Stop	спиране на процес

! Сигналите SIGKILL и SIGSTOP не могат да бъдат хващани, блокирани или игнорирани.

Сигнали до 16 номер са стандартни - те са още от зората на Unix. Тия от 17 номер нагоре са с различни в различните версии на UNIX.

3) int pause(void) - чака сигнал; винаги връща -1.

4) unsigned int alarm(unsigned int seconds) - планира изпращането на SIGALRM

бр. секунди, след изтичането на които ще се изпрати сигналът SIGALRM на текущия процес

Връщата стойност на alarm():
В общия случай alarm() връща оставащия брой секунди до изпълнението на предходна функция alarm() (при извикване на нов alarm() старият брояч се нулира, броенето започва отначало, т.е. сигналите не се натрупват)

```
alarm(10)
    ⋮ 2sec
    ⋮
alarm(5)   // връща ст-ст 8
```

Обикновено в една програма се извиква еднократно alarm() => върнатата му стойност е 0.

19.11.2024г.

Тема 13: Разговори между процеси чрез socket(7): socket(2), connect(2), bind(2), listen(2), accept(2).

Това е още един вид връзка между процесите. Досега сме дискутирали
- pipe (анонимна тръба)
- mkfifo (именувана тръба)

Да си припомним:

1) Анонимна тръба
Създава се от един процес, който вижда и края й за писане, и края й за четене, и който я подава на своите наследници. Обикновено един роднински процес използва края за четене, а друг от роднинските процеси - края за писане.
+ „Хубавото" на тази тръба е, че знаем кои процеси сборува.
- Проблемът е, че тази тръба е достъпна само за роднински процеси. Един (родителски) процес трябва да се погрижи както за конструирането на тръбата, така и за създаването на процес(и) дете (деца).

2) Именувана тръба
+ един процес я създава с mkfifo(), дава й права и всеки процес, който има достъп, може да я използва някой от краищата й - за четене или писане, т.е. да праща или изпраща байтове по тази тръба.
- Проблемът е, че не знаем кои процеси я ползват, кои използват единия край и кои другия. Всички с достъп могат да я използват
права за

3) Сокет
Инструмент, който създава връзка м/у два процеса (концепцията сокет, а не системното извикване socket()).
+осигуряване на процесите възможност за разговаряне през интернет
Тази връзка е малко по-сложна от тръбата - двустранна е, а обикновено тръбите (съществуващи още от началото на UNIX) са едностранни.
Всяка разполагане с две тръби  едновременно.

двупосочна конекция

Концепцията покрива не само тези връзки, които осигуряват потоци.

! Целта е да се сборуват нероднински програми => трябва по някаков начин да се идентифицират (да има идентификация поне на единия компютър) и затова се използва пр-вото на имената.
Поне единият процес трябва да се именува, така че другите процеси да го открият и въобще да започне някакъв „разговор" между процесите.
Поради тази причина има няколко системни извиквания, които се изпълняват в определена последователност (виж схема 1).
Процесът, който създава име, наричаме сървър.
Другия наричаме клиент.
И при двата процеса първата от действия започва със системната функция socket().

Думата socket не е случайно избрана, има следните семантики:
- гнездо на птица
- гнездо за сваряване на жици

Обяснението е, че socket() създава крайна точка за комуникация. И двата процеса (и клиентът, и сървърът) създават такова „гнездо" със съответните параметри (3бора - последният може да е без значение и да се пропуска в повечето случаи, първите 2 са по-важни):

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

domain — описва област (пр-во на имената), в която ще работим, т.е. как ще идентифицираме процесите

type — тип на връзката

protocol — номер на протокола

I) Параметърът domain има много стойности, но ще разгледаме само някои от тях:

- AF_UNIX, AF_LOCAL: връзката ще е локална (на едно устройство); нещо, което ще получи „гнездото" на сървъра, ще бъде псевдофайл от тип socket във FS-та.
- AF_INET: IPv4 - IP адресът се състои от 4B
- AF_INET6: IPv6 - IP адресът се състои от 6B
- :

Много от тези интерфейси са на големи компании, които са създали собствени стандарти:
AF_IPX - на Novell
AF_APPLETALK - на Apple
AF_DECnet - на DEC и т.н.

Именуването на процесите е различно в различните области

II) Параметърът type също има няколко разновидности

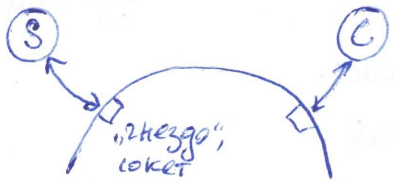
- SOCK_STREAM: най-често използваната; конекция се изгражда м/у два процеса, когато в двете посоки текат байтове
- SOCK_DGRAM: без конекция; отделни пакети се изпращат м/у процесите: поредица от пакети, а не от байтове! → цели масиви от байтове които имат смисъл на интернет пакети
- SOCK_SEQPACKET
- SOCK_RAW
- SOCK_RDM
- SOCK_PACKET: не бива да се използва в нови програми

! По-модерните варианти на 2тих поддържат допълнителни битове към type
O_NONBLOCK, O_CLOEXEC

III) Параметърът protocol задава конкретният протокол, но е възможно и да не се подаде. За някои типове дейности в интернет е нужно да се подаде

Повече ще говорим за сокети, които са отворени с типа SOCK_STREAM
=> Двупосочен байтов поток

Към момента:



S
sfd = socket(...)

C
cfd = socket(...)

2) Следващата стъпка, която сървърът предприема, е даването на име на fd-ия.

bind (sfd, name)

т.е. сървърът си именува „гнездото", като в зависимост от областта (domain-а) името има различна структура

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
присвоява име на sockfd

дължина на структурата

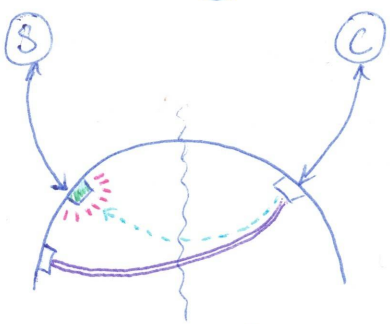
Структурите в различните области са различни, респективно имат различни дължини.

S
sfd = socket(...)
bind (sfd, name, ...)

C
cfd = socket(...)

ако областта тук е интернет, name в долната функция е IP адрес + порт
някои номера на портове са запазени

Сървърът и клиентът са процеси, които изпълняват следните команди в точно определена последователност, за да изградят връзка помежду си:



S
1) sfd = socket(...)
2) bind (sfd, sname)
3) listen (sfd, backlog)
5) fd = accept (sfd, cname)
fork() - само при fork сървър

C
1) cfd = socket(...)
4) connect (cfd, sname

схема 1: договаряне и установяване на връзка

3) listen(?) - „слуша" за постъпващи заявки

int listen(int sockfd, int backlog);

Маркира сокета, който е sockfd, като listener („подслушващ").

I параметър: sockfd е файлов дескриптор, който ни връща socket() или SOCK_STREAM или SOCK_SEQPACKET.

II параметър: backlog дефинира дължината на опашката за постъпващи заявки за свързване.

! Ако не се извършва периодично accept(), може да се натрупат няколко заявки от клиенти в опашка. Така, ако сървърът е зает с друга дейност в момента и не може да изпълни accept(), заявката отива в тази опашка

4) connect() - клиентът изпраща заявка за свързване
5) accept() - приема заявка и изгражда конекция
Новата връзка трябва да се построи в нов fd => fd accept()
върнатата ст-ст е файлов дескриптор

09.12.2021г.

Тема 16: Споделена памет shm_overview(7): shm_open(3), ftruncate(2), mmap(2)

Теми 16, 17 и 18 са посветени на споделени обекти

До момента сме разглеждали fd-и: които сочат към обект за обмен на информация м/у процеси и някакъв друг обект. Чрез тях четем и пишем данни с read() и write().

Реализацията на fd-ите (тяхната синхронизация, осъществяването на връзката с другия обект, поддържането на комуникацията) е изцяло в ядрото. Програмистът използва наготово функционалностите на fd-ите за четене и писане в комуникационните канали

(fd може да сочи към отворен файл, към тръба (pipe / fifo), към конекция към друг процес (socket))

Когато обаче задачата на програмиста не е свързана само с обмен на данни и се изисква и друг вид комуникация м/у процесите, по схема, неизпълнима от стандартните, завършени функционално комуникационни канали от високо ниво, се използват примитиви от по-ниско ниво и подреждане на процесите във времето при работата им с общите ресурси.

Стандартът POSIX предлага такива инструменти

Двата инструмента са еквивалентни (1) може да се реализира чрез 2) и обратното)

1) споделена памет + синхронизиране действията с нея чрез семафори
2) съобщенията

В теорията на конкурентното програмиране това са основните модели за комуникация м/у процесите (+ още един модел: отдалечено изпълнение на процедури).

Въпросните два най-често използвани механизми (1) и 2)) за взаимодействие м/у процеси имат реализация в UNIX, в по-старият стандарт IPC (Inter Process Communication), който възниква в System 5.

Тема 16: Как няколко процеса използват споделена памет:

Това се осъществява най-вече със следните 3 инструмента (системни извиквания):

1) shm_open() за създаване (отваряне) на споделена памет

тя трябва да се дефинира като обект, а за да се използва от процесите, те трябва да я „забележат", да я намерят => работим в пр-вото на имената.

2) ftruncate() за оразмеряване на споделената памет (т.к. в shm_open() не присвоява параметър за размер на спод. памет)

3) mmap() за присъединяване споделената памет към адресното пр-во на процеса

* ftruncate() може да се използва и за файлове
  - ако намалява размера на файла, губим данни
  - ако увеличава размера на файла, се добавят виртуални нули

Забележка: За разлика от open() за отваряне на файл, функцията shm_open() НЕ използва цялото пр-во на имената на компютъра, а само една абстрактна, виртуална директория. В Linux те са в конкретна директория - /dev/shm. Но това е конкретно за Linux. Иначе в стандарта POSIX такова директория не е стандартизирана - тя си е отделна виртуална подсистема, виртуална директория, а не е някакво пространство на твърд диск.

Сега ще разгледаме по-обстойно всяко от изброените по-горе системни извиквания:

1) int shm_open(const char *name, int oflag, mode_t mode)

I параметър: име - не е пълно име на файл като при open(), а само стринг

II параметър: флаг(ове) - задава начина на ползване на обекта

III параметър: mode - права за достъп (~~задава се само при O_CREAT за новообект~~) задължителен параметър, но се взема предвид само при вдигнат флаг O_CREAT

ФЛАГОВЕ

O_RDONLY : отваря паметта само за четене

O_RDWR : отваря паметта за четене и писане

O_CREAT : ако паметта не същ., се създава

O_EXCL : заедно с O_CREAT; ако паметта вече същ., връща грешка EEXIST

O_TRUNC : ако паметта същ., нулира размера ѝ

Върнатата ст-ст на shm_open е integer като fd, но не е такъв fd, който може да се използва за работа със споделената памет, а просто ни дава достъп до структурата, която ядрото поддържа и с която ще извършваме операциите за манипулиране на споделената памет.

2) int ftruncate(int fd, off_t length)

Новата споделена памет (при създаване с O_CREAT) и вече съществуващата, но отворена с флаг O_TRUNC, имат нулев размер.

Преоразмеряването се извършва с ftruncate().

при файлове:

- ако задава по-малък размер:

„реже"

- ако задава по-голям размер:

Добавя нови мета данни във файловата таблица, които ще окажат, че има още сектори (към момента ще са виртуални, но ще се създадат при нужда).

при спод. памет:

При прилагане към споделена памет ftruncate() ще предизвика създаване на сегмент, парче от АТ (адресна таблица), която таблица ще дефинира група страници в паметта. Тяхният размер (малък) е достатъчен, за да побере данните, които ще ползваме. Както и при работата с файлове, тези страници няма да бъдат заделени веднага, а само ще бъдат описани като бройка, и ще бъдат предоставени при нужда - чак когато някой от процесите започне да използва въпросната памет. Това ядрото ще изисква да се предоставят тези

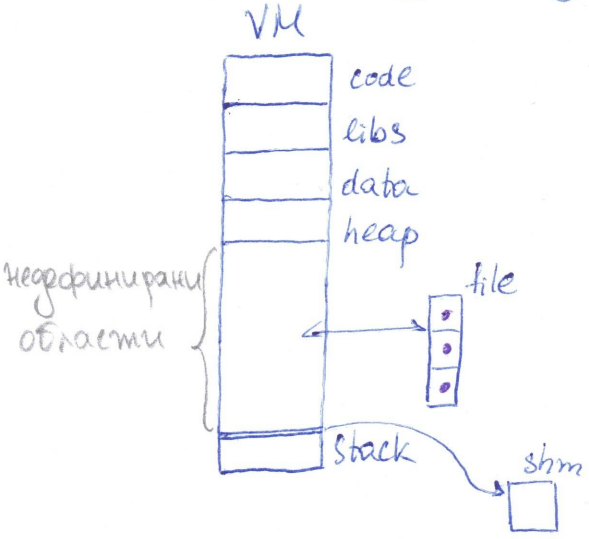
3) void * mmap(void* addr, size_t length, int prot, int flags, int fd, off_t offset)

fd обр. към файл или към споделена памет

Смисълът на тази операция е масивът от байтове, представен от fd-ия, да бъде вкаран във VM (виртуалната памет) на процеса.

- Когато работим с файл, се задава откъде (от кой байт на файла) и каква дължина на файла ще бъде приведен(а)
- Когато работим със споделена памет, обикновено offset е 0, тъй като цялата споделена памет ще се приведе(и).

Виртуалната памет (VM) е като един (безкраен) масив от байтове, които са видими за процеса. VM е разделена на страници - някои страници не са дефинирани и са празни, а други са дефинирани. В края на това адресно пр-во е стекът на програмата (тук се пазят извиквания на функции или локалните им променливи - изобщо данни, свързани с управлението на програмата). В началото е кодът на програмата, който се изпълнява, после може да има библиотеки, статични данни, динамични данни (heap). Но има и огромни празни места



С mmap() вкарваме някъде в тези недефинирани области в локалната памет ~~споделен обект~~ file.

Този интервал от файла са сектори, страници от паметта (локалната). Тези сектори се зареждат в RAM паметта и се включват във VM на процеса.

Във file можем да четем и пишем с нормални операции за адресация. Когато затворим map-инга, промените ще се отразят във файла.

Секторите от файла не се зареждат веднага в RAM паметта, ами във VM на процеса, в неговата PT, се създават (дефинират) нови страници със свойството load-on-demand.

Аналогична е ситуацията, когато не работим с файл, а със споделена памет.

05.12.2024г.

Тема 17: Семафори sem_overview(7): sem_init(3), sem_post(3), sem_wait(3), sem_destroy(3)

Ако няколко процеса ползват споделена памет, трябва да я ползват в правилен ред, за да не възникне race condition. Налага се да имаме група семафори, броячи и други променливи, които да регулират достъпа до данни във времето. => В стандарта POSIX трябва да има и семафори. В тази лекция се разглежда група от системни извиквания за семафори.

Основната страница, която описва работата със семафорите, е sem_overview

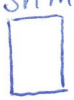
! Ние като потребители не можем сами да си направим семафори поради липсата на достъп до примитиви като block(), wakeup(pid) - не можем да приспиваме или събуждаме процеси, това е дело на ядрото, а въпросните примитиви са скрити някъде надълбоко". Самата реализация на семафорите е скрита в ядрото и то осигурява тези услуги.

Семафорите биват 2 вида → named
→ unnamed

↳ едните се създават в пр-вото на именатa (пак в някаква „тайна" директория)

функции: sem_open() за създаване на такъв семафор; връща у-тел към структура, която обслужва семафора
Такъв семафор трябва да се затвори накрая и да се изтрие от процеса, който го е създал, със sem_unlink().

↳ неименуван (анонимен), абстрактен семафор
Няма име => за да е видим за процесите, се слага в споделената памет.



В споделената памет shm може да се създават обекти семафори.

Анонимният семафор се инициира със sem_init() и после се унищожава със sem_destroy().

! Семафорът е обект, а от ООП знаем, че всеки обект следва да има конструктор и деструктор. Конструкторът на семафора е sem_init(), а деструкторът му е sem_destroy().

		named	unnamed
+	констр.	sem_open()	sem_init()
-	деструк.	sem_unlink()	sem_destroy()

Да разгледаме поотделно всяко от системните извиквания:

1) sem_open() - няма пълна аналогия с open() за файлове

sem_t *sem_open(const char *name, int oflag [, mode_t mode, unsigned int value]) - създава и инициализира (отваря) семафор

oflag:

O_CREAT - създава семафор

O_EXCL - заедно с O_CREAT; ако семафорът същ., връща грешка EEXIST

sem_open() създава именуван семафор, задава флагове за начина на работа със семафора, права за достъп и начална стойност. - Това е конструкторът на семафора. А тъй като всичките процеси, които ще ползват семафора като споделен обект, се обръщат към него само с името му

2) sem_init() - за създаване на неименуван семафор

int sem_init(sem_t *sem, int pshared, unsigned int value);

↓ трябва да имаме готова структура за семафор

↓ памет

↓ начална ст-ст (*) /началната ст-ст на брояча, ≥0/

Аргументът pshared индикира дали семафорът да бъде споделен между нишки на един процес или между процеси.

- ако pshared е 0, семафорът бива споделен м/у нишки на един процес и трябва да бъде локализиран на такъв адрес, видим от всички нишки (т.е. да е глобална променлива или да е поместен в heap-а).
- ако pshared е нещо, семафорът бива споделен м/у отделни процеси и трябва да бъде разположен в споделената за процесите памет.

(*) показва колко процеса или нишки могат да преминат бариерата „охранявана" от семафора.

1) и 2) са функциите за създаване на семафора

3) sem_post() - функцията, която увеличава брояча с 1-ца

int sem_post(sem_t *sem);

Ако има приспани процеси, един от тях се събужда
Това е силен семафор - събужда се най-отдавна приспаният процес
В противен случай, ако семафорът бъде сам, дори при най-прости задачи за синхронизация може да предизвика гладуване.

! sem_post() никога не блокира, просто увеличава брояча и изпълнението на програмата продължава. Уе може да предизвика приспиване на процеси. Но може, разбира се, да настъпи грешка OVERFLOW: броячът на семафора е от тип int, има горна граница. При препълване на брояча възниква тази грешка ... фатална грешка ☺.

4) sem_wait() - има няколко, различни варианта, за да предоставят по-голяма функционалност.

int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);

- sem_wait() - най-простата форма; намалява брояча на семафора с 1-ца ако броячът е ≤0, процесът ще бъде приспан и охранен в структурата, която семафорът използува за приспаните процеси

Така приспаният процес може да бъде приспан за неопределено дълго време, т.е. sem_wait() е БАВНО СИСТЕМНО ИЗВИКВАНЕ.

4) Когато процес бива приспан, трябва да се запази информацията за това коя структура е предизвикала приспиването му. Тази информация се пази в ЯДРОТО.
Ако кой приспаящия процес се изпрати сигнал, ядрото трябва да достъпи вътрешната структура, да я възложи да събуди процеса => АВАРИЙНО СЪБУЖДАНЕ на процеса. Тогава се връща грешка EINTR. Накратко, сюжетното действие: Процесът е приспан, но кой него пристига сигнал и трябва да се обработи. Процесът се събужда аварийно и операцията sem_wait() завършва неуспешно.

- int sem_trywait(sem_t* sem) - намалява стойността на брояча с 1; процесът не се блокира, а се връща грешка EAGAIN

- int sem_timed_wait() - времево приспиване
Реализиран е със struct timespec {
time_t tv_sec; → задава определен
long tv_nsec; времеви интервал
};

5) sem_destroy() - унищожава неименуван семафор

17.12.2024г.

Тема 18: Съобщения mq_overview(7): mq_open(3), mq_send(3), mq_receive(3)

Този инструмент е независим от споделените памети и семафорите. Задача за синхронизация може да се реши и само чрез изпращане на „писма" м/у процеси. По-нататък е представена идея за доказателство защо двата механизма (семафори и изпращане на съобщения) са еквивалентни. В някои случаи по-удобен е единият механизъм, а в други случаи - другият.

В тази лекция се разглеждат следните 3 системни извиквания:

1) mq_open() - създава нова или отваря вече Съществуваща опашка за съобщения („пощенска кутия")

Опашките за съобщения са споделени обекти и трябва да са достъпни за група процеси. Достъпът се осъществява чрез имената на опашките. Т.е. работим в пр-во на имена" - имената на Съществуващите пощенски кутии се съхраняват в някоя директория, както е при именуваните семафори и споделените памети.

При отваряне на Съществуваща пощенска кутия: намираме я по името ѝ и указваме флагове. За създаване на нова пощенска кутия освен задължителните и флагове, с които тя ще се полива, но и права за достъп и структура, в която се описват свойствата на пощенската кутия

```
mq_open(const char *name, int oflag);                    за стара кутия
mq_open(const char* name, int oflag, mode_t mode, struct mq_attr *attr);
                                                          за нова кутия
```

ФЛАГОВЕ:

O_RDONLY : отваря опашката само за четене
O_WRONLY : отваря опашката само за писане
O_RDWR : отваря опашката за четене и писане
O_NONBLOCK: отваря опашката в асинхронен режим (процесът не се блокира, операциите завършват с грешка EAGAIN).
O_CREAT: ако опашката не същ., се създава
O_EXCL: заедно с O_CREAT; ако опашката същ., връща грешка EEXIST

СТРУКТУРАТА:

```
struct mq_attr{
    long mq_flags;   // флагове: 0 или O_NONBLOCK
    long mq_maxmsg;  // максимален брой съобщения в опашката
    long mq_msgsize; // максимален размер на съобщение (в байтове)
    long mq_curmsg;  // брой съобщения в опашката
};
```

2) mq_send() - изпраща съобщение

```
mqd_t mq_send(mqd_t mqdes, const char* msg_ptr, size_t msg_len, unsigned msg_prio);
```

Грешки, свързани с асинхронния начин на употреба:

ETIMEDOUT
EAGAIN
EINVAL

3) mq_receive() - за получаване на съобщение; изважда първото от пощенската кутия (! Взема най-старото съобщение от тези с най-висок приоритет).

Ако пощенската кутия е празна, процесът се приспива (ако е в синхронен режим на работа).

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned *msg_prio)

1), 2) и 3) са БАВНИ СИСТЕМНИ ИЗВИКВАНИЯ => процесът може да бъде приспан за по-дълго време.; грешка EINTR

Стандартна памет със семафори ≡ пощенски кутии (опашка за съобщ.)
(еквив.)

Ще покажем, че двата подхода са еквивалентни, ако инструментите от единия механизъм могат да се интерпретират с инструментите на другия механизъм.

(=>) Опашката без приоритети просто замисляне с опашка с приоритети

(<=) Ще направим опашка с по-голям размер, т.е. с повече на брой места за съобщения, но съобщенията ще са с нулев размер - ще са празни съобщения.

Тук ще интерпретира работата на семафорите. Да я кръстим semq
Всички съобщения ще са с нулев приоритет

```
semq
  sem_init(cnt0)
    sem_open(semq, ...)
    for i=1 to cnt0
      sq_send(semq, ...)

sem_wait
  sq_receive(semq, ...)

sem_signal
  sq_send(semq, ...)
```