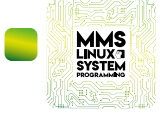
Системно програмиране за Линукс

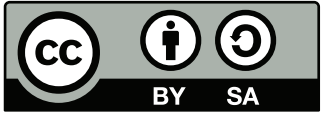
Средства и механизми за междупроцесна комуникация.

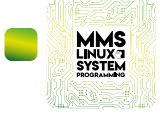
Ангел Чолаков

MMSolutions
A ThunderSoft Company

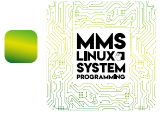
02.06.2021г.





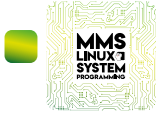


Съдържание I



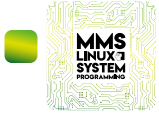
Цел на презентацията

- Да опита да поясни:
  - примерни сценарии и предпоставки за обмен на данни между задачи и потоци;
  - какви средства предоставя ОС за междупроцесна комуникация;
  - как описаните примитиви се категоризират според целевата функция;
  - какви политики и механизми се ползват за подсигуряване на междупроцесна комуникация;
  - илюстрира практически примери в контекста на Posix Pthread API и Линукс ОС



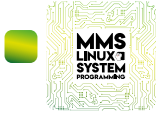
Предпоставки за междупроцесна комуникация

- Очертават се:
  - **обмен на пакети работни данни** за нуждите на кооперирващи се процеси, развиващи с в общо виртуално адресно пространство върху една система;
  - отправяне на **заявка за изпълнение на услуга** от страна на операционната система;
  - **сигнализация** за настъпило **събитие** или **изключителна ситуация** по време на работа;
  - **поточен синхронизиран** или **асинхронен обмен на пакети данни** с вариращ размер между различни компютърни системи, свързани в мрежа

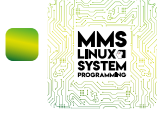


Опит за класификация на подходите

- Възоснова на:
  - употреба на **специализирани комуникационни похвати и средства**, често пъти включващи запис и четене на данни към и от ядрото на ОС в ролята на посредник и арбитър;
  - дефиниране и заделяне на **споделени региони памет** в общото виртуално адресно пространство без необходимост от междинни обръщения към ОС и допълнителни системни извиквания

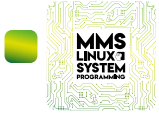


Фокусът в настоящата презентация се поставя върху методи за обмен на данни между процесите в рамките на една компютърна система без разглеждане на мрежови примитиви и парадигми.



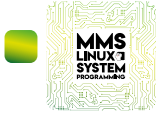
Средства за комуникация: подкатегории

- Под контрола на ОС се обособяват:
  1. **буферирани** или **не**, **блокиращи** или **не**, **поточно (byte stream)** ориентирани канали за предаване на информация;
  2. методи, базирани на **комуникация с работни обекти (съобщения)** и привързване на процесите към опашки от входящи и изходящи съобщения **(message queues и sockets)**;
  3. технология с **неявна вътрешна синхронизация** между асоциираните процеси или такава, изискваща **експлицитна допълнителна синхронизация**;
  4. методи с **изчерпващ характер на четенето** на нови данни или такива, при които четенето е разрешено за множество процеси или потоци без то да е свързано с безвъзвратно унищожаване на прочитаните данни и изваждането им от комуникационната среда



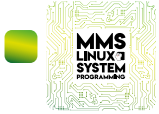
Средства за комуникация: продължение

- Примери към посочените подкатегории:
  - по т.1 - вече разгледаните на упражнения pipes (програмни тръби);
  - по т.2 - обекти съобщения и опашки от съобщения (message queues);
  - по т.3 - операции по четене и запис чрез файлове и файлови дескриптори (автоматично синхронизирани) и комуникация чрез споделени обекти или блокове памет (налагащи явна координация);
  - по т.4 - извличане на количество прочетени байтове от файлов дескриптор (данните се източват от програмния канал и вече не са достъпни) или четене от споделен масив структури в паметта (обектите не се освобождават в резултат на четенето)

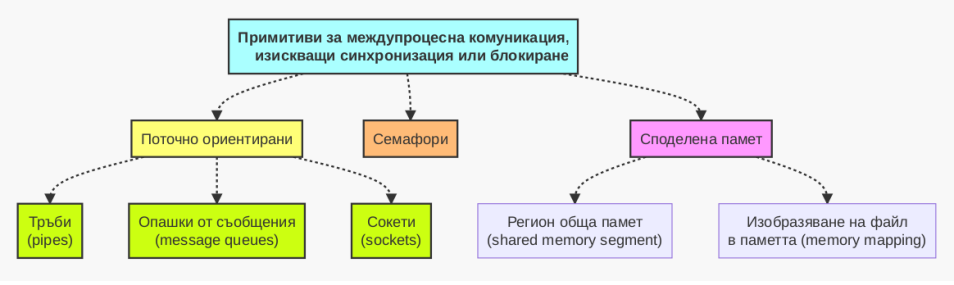


Междупроцесна комуникация: разновидности механизми

- Съществуват два основни вида - посредством:
  - **сигнали** - обслужват се посредством подсистема в ядрото на ОС служат за основна **форма на междупроцесна нотификация**;
  - **програмни канали** - служат за обмен на порции данни с произволен размер
- Програмните канали се разделят допълнително на:
  - **поточни**: **тръби (pipes)**, **опашки от съобщения (message queues)** и **сокети (sockets)**;
  - решения със **споделена памет**: чрез маркирани **сегменти за целта (Posix shared memory)** или **изобразяване** на обекти в споделени сегменти памет **(memory mapping)**

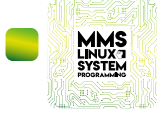


Изискващи синхронизация видове примитиви: диаграма



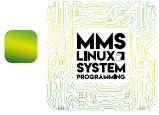
pic. based on `https://man7.org/conf/lca2013/IPC_Overview-LCA-2013-printable.pdf`

А как изглежда програмният интерфейс за ползване на комуникационни обекти и инструменти в контекста на Линукс ОС и Posix libpthread?

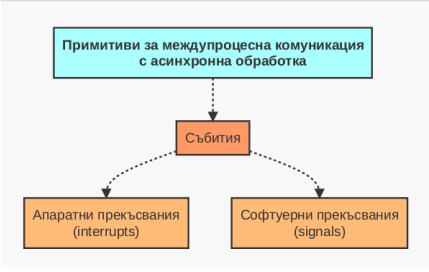


Работа със сигнали

- Дефиниция и предназначение:
  - служат за **известяване** на процесите за възникнали **събития**, представящи изключителни състояния или заявка за програмно взаимодействие;
  - диспечерират се от ядрото и се обслужват посредством структури от данни, част от контролния блок за управление на процесите;
  - не са проектирани и не са пригодни за разширена, максимално ефикасна или надеждна форма на комуникация, а служат по-скоро само като **базов способ за нотификация**



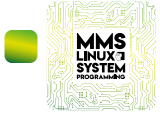
Разновидности асинхронни комуникационни средства



pic. based on `https://en.wikipedia.org/wiki/Interrupt_handler`

Реализация на сигналите в Линукс ОС

- Разчита на:
  - **побитови полета**, част от контролния блок за управление на процесите (представен чрез **task_struct** структурата, описваща атрибутите на всеки процес);
  - възникването на събитие и реалното му обработване се отразява чрез членове от специализиран тип **sigset_t saved_sigmask** и `struct` **signal_struct *signal** инстанции, с маска на временно блокираните сигнали в **blocked** в контекста на Линукс ОС;
  - посочените структури идентифицират всяко събитие чрез **цяло число** и **адрес на функция за изпълнение при възникване**;
  - арбитраж и реално вътрешно обслужване посредством ядрото на Линукс ОС;
  - ако разработчикът не укаже програмно функция за обслужване на събитие по сигнал, ядрото изпълнява стандартна такава по подразбиране

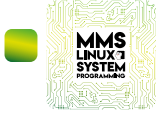


Последователност по обработка на сигналите

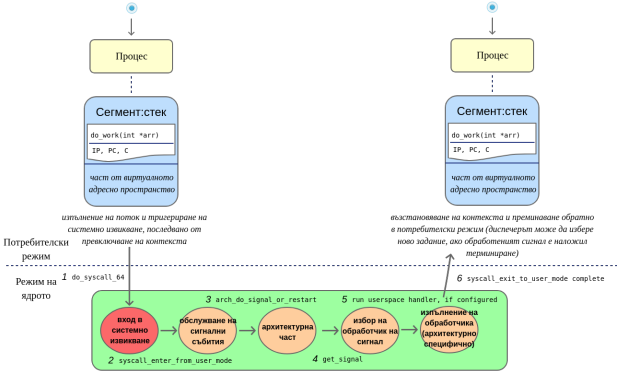
- Базирана е върху:
  - ако даден сигнал не е временно забранен и процес се намира в чакащо състояние, възникването на събитието е съпроводено с евентуално събуждане на процеса и поставянето му в опашката на готовите задачи;
  - ако сигнал възникне по време на изпълняващ се процес, последният бива прекъсван и контролът се предава на ядрото;
  - при активация от страна на диспечера и след всяко изпълнение на системно извикване се прави проверка на флаговете в **signal** and **blocked** полетата, за да се доставят и обслужат неотчетени сигнали;
  - **подпрограма за обслужване на събитието** се указва поредством инстанция на **sigaction структура**;
  - преди изпълнение на обработчика на събитието - ядрото съхранява състоянието на прекъснатия процес, за да може контекстът да бъде възобновен по-късно;
  - контролът се връща обратно на потребителското задание, ако то не се терминира, посредством опционално имплементирано системно извикване: **sigreturn**, а ако такова отсъства - по архитектурно специфичен начин

Механика на сигналното взаимодействие: продължение

- Особености:
  - само ядрото и суперпотребителите могат да разпращат сигнали до всички процеси;
  - нормален потребителски процес е в състояние да сигнализира само процеси с позволени потребителски или групов идентификатор или със съвпадащ групов идентификатор;
  - с изключение на сигналите **SIGSTOP** and **SIGKILL** процес може програмно да заяви блокиране на всички останали сигнали, като в този случай блокираните остават в очакващо състояние;
  - всеки процес разполага с **побитов вектор (битова маска)**, която разкрива кои сигнали са позволени



Обслужване на сигнали: диаграма

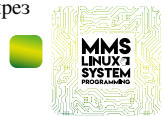


Опростена последователност в процеса на изпълнение на системно извикване и обработване на възникнали сигнални събития от Линукс ядрото (в контекста на x86_64)

pic. based on https://elixir.bootlin.com/linux/v5.12.1/source/arch/x86/kernel/signal.c#L707

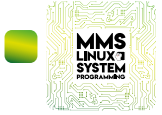
Обработка на сигнали в процес с подчинени потоци

- Специфики:
  - **породен процес потомък** или нишка **наследяват** от родителския процес битовата маска с разрешени за детекция сигнали;
  - всеки поток разполага със **собствен контролен блок за диспечериране**, който съдържа независимо конфигурируема битова маска за контрол върху набора позволени сигнали;
  - дадено събитие може да е съпроводено със сигнал, насочен към процес родител или към конкретна нишка (като резултат от опит за неправомерен достъп до паметта - **SIGSEGV** или грешка при изчисления - **SIGFPE** например);
  - при процес с няколко активни потока и при условие, че те не са забранили отработването на определен сигнал, диспечерът на ядрото избира към кой от потоците да бъде предаден сигналът, насочен към родителския процес;
  - всяка нишка указва своето терминиране или спонтанно възникнала изключителна ситуация чрез генериране на специализиран сигнал **SIGCHLD**



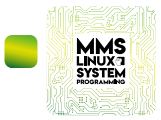
Прихващане на сигналите в Линукс: приложна секция

- Полезни API функции:
  - `int sigprocmask(int how, const sigset_t * restrict set, sigset_t * restrict oldset);` - служи за промяна на списъка с блокирани сигнали;
  - `int sigpending( sigset_t *set );` - разкрива кои сигнали са позволени и кои чакащи;
  - `int sigsuspend(const sigset_t *mask);` - преустановява изпълнението на процес до възникване на определен сигнал;
  - `typedef void (*sighandler_t)(int);` - прототип на обработваща даден сигнал функция;
  - `sighandler_t signal (int sig, sighandler_t handler);` - асоцииране на сигнал с обработваща функция



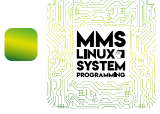
Шаблон за програмиране на обработваща функция

```c
/* when the parent process wants to ignore the SIGCHLD signal */
signal(SIGCHLD, SIG_IGN);

/* or when the parent process wants to handle the SIGCHLD signal explicitly */
struct sigaction action;
sigemptyset(&action.sa_mask );
action.sa_handler = my_handler;
action.sa_flags = SA_NOCLDWAIT;
sigaction(SIGCHLD, &action, NULL);

/* where the signal handler has been defined beforehand */
static void my_handler(int signum){
        /* some reentrant code follows here */
}
```



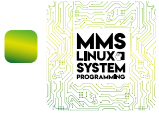
Работа със сигнали: разглеждане на пример:

`https://man7.org/tlpi/code/online/dist/signals/sigmask_longjmp.c.html`



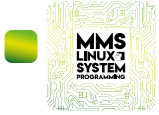
Именувани програмни канали (fifos)

- Дефиниция:
  - представляват **форма на еднопосочен канал на комуникация** тип ”пощенска кутия” и позволяват пренасочване на изходния поток данни от един процес да служи за входен поток на друг;
  - за процесите **вътрешната реализация** на комуникационните тръби е **прозрачна** и тя няма отношение върху поведението им

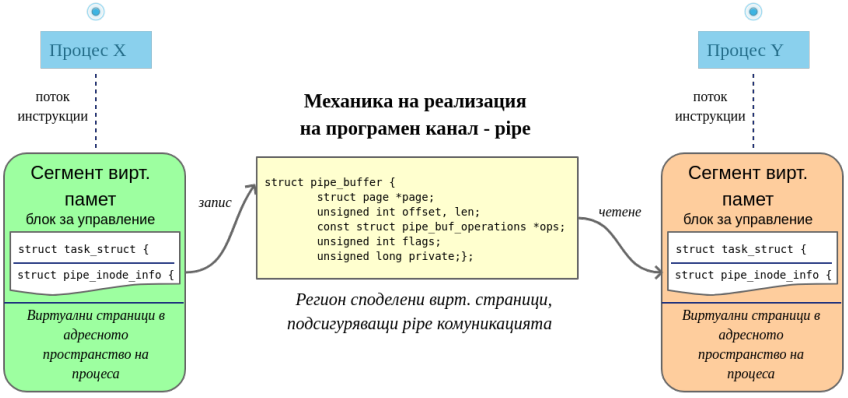


Програмни канали: имплементация в Линукс

- Базира се на:
  - **тръбите** се изграждат чрез **двойка файлови дескриптори**, насочени към временно споделена обща странична референция в адресното пространство на виртуалната памет;
  - всяка инстанция от двойката дескриптори съдържа **указатели към различни привързани процедури** - за запис от едната страна на тръбата и четене - от другата;
  - при запис потокът от байтове се копира в общия страничен регион, а при четене - се прави копие от споделения блок към сегментите на процеса читател;
  - **вътрешната синхронизация** и **контрол на състоянието на процесите** става от ядрото, като то гарантира прилагане на обекти ключалки, опашки, блокиране на процеси при запълване на тръбата с разпращане на сигнали



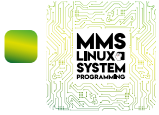
Posix named pipes: илюстрация на механизма



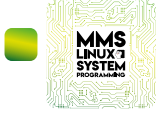
pic. based on `https://elixir.bootlin.com/linux/latest/source/include/linux/pipe_fs_i.h#L26`

Posix fifos: приложна секция

- Полезни API функции:
  - `int` mkfifo(`const char` *pathname, mode_t mode); - изгражда програмна именувана тръба посредством предоставен аргумент - път към файлов обект;
  - `int` mkfifoat(`int` dirfd , `const char` *pathname, mode_t mode); - аналог на горната функция, но с разлики в третирането на аргумента път и неговия релативен или абсолютен характер

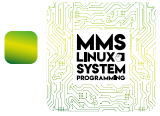


Работа с програмни тръби: разглеждане на пример:
`https://man7.org/tlpi/code/online/dist/pipes/fifo_seqnum_client.c.html`



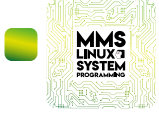
Опашки от съобщения: (message queues)

- Дефиниция на подхода:
  - процесите обменят данни под формата на **съобщения**, пакетирани в структури със съответен тип и атрибути;
  - съобщенията се организират в **привързани към процесите опашки**, всяка от които разполага с уникален идентификатор;
  - изпращането и получаването на съобщения се реализира с помощта на **системни извиквания** и таблица на създадените и активни опашки в ядрото на ОС;
  - механизмът работи с **копиране на данни** от адресно пространство на един процес в пространството на друг;
  - опит за изпращане в пълна опашка или за четене от празна води до **блокиране** на процеса писател или читател съответно

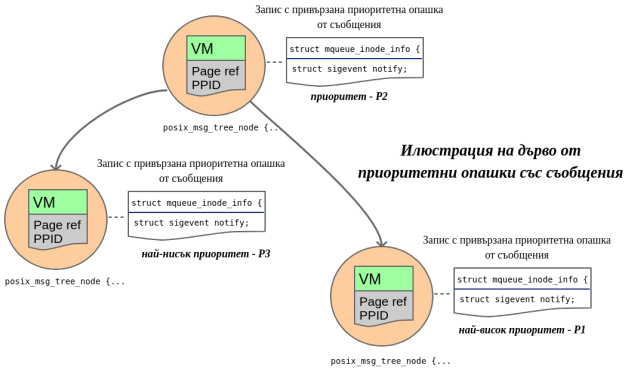


Опашки от съобщения: реализация в Линукс

- Основава се на:
  - създаването на опашка е асоциирано със заделяне на свързан файлов дескриптор от тип **mqd_t**;
  - този дескриптор е насочен към логически обект (инстанция на `struct` **mqueue_inode_info**, съдържаща **червено-черно дърво** от елементи съобщения), достъпван чрез виртуална файлова система;
  - Линукс третира всеки файлов запис опашка като традиционен файлов обект и позволява мониторинг с обичайните системни извиквания (като **select** и **epoll** например);
  - всяко разпращано съобщение притежава **атрибути** и **указан приоритет**, като първи се доставят високоприоритетните съобщения;
  - веднъж изградена, една опашка от съобщения остава да консумира памет (структури на ядрото и страници памет), докато не бъде освободена чрез **mq_unlink**



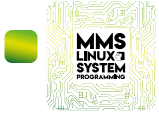
Диаграма на вътрешното представяне



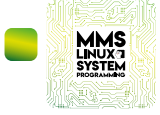
pic. based on `https://man7.org/linux/man-pages/man3/mq_notify.3.html`

Posix message queues: приложна секция

- Полезни API функции:
  - mqd_t mq_open(`const char` *name, `int` oflag, mode_t mode, `struct` mq_attr *attr); - създава или отваря нова Posix опашка от съобщения;
  - `int` mq_send(mqd_t mqdes, `const char` *msg_ptr, size_t msg_len, `unsigned int` msg_prio); - добавя съобщение за разпращане в опашката, идентифицирана от подадения дескриптор;
  - ssize_t mq_receive(mqd_t mqdes, `char` *msg_ptr, size_t msg_len, `unsigned int` *msg_prio); - извежда най-старото постъпило съобщение с най-висок приоритет от указаната опашка;
  - `int` mq_notify(mqd_t mqdes, `const struct` sigevent *sevp); - позволява извикващият процес да се обвърже с доставянето на съобщения по указаната опашка;
  - `int` mq_unlink(`const char` *name); - заявява последващо освобождаване на посочения обект опашка, когато всички ползватели затворят референциите към съответния дескриптор обект;
  - `int` mq_close(mqd_t mqdes); - затваряне на файловия дескриптор, идентифициращ опашката

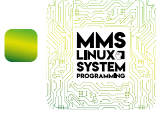


Работа с Posix опашки от съобщения: разглеждане на пример:
`https://man7.org/tlpi/code/online/dist/pmsg/pmsg_receive.c.html`

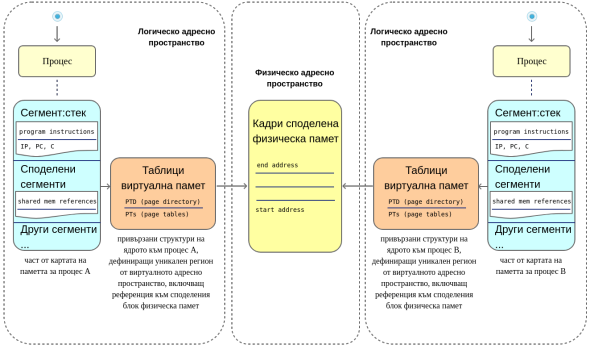


Механизъм на комуникация с обща памет

- Дефиниция:
  - наподобява размяна на съобщения, но **без необходимост от копиране на данни** между адресните пространства на процесите и/или процесите и ядрото;
  - позволява множество процеси да изобразяват сегменти от виртуалното си адресно пространство в **общ физически сегмент на оперативната памет**;
  - възможен е **директен обмен на данни** между процесите, но с добавен ангажимент към разработчиците за обезпечаване на синхронизация и интегритет при организиране на комуникацията;
  - механизмът разчита на подсистемата за **странично-сегментна организация** на виртуалната памет



Работа с обща памет: опростена диаграма

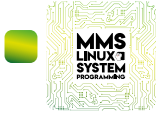


Схематично представяне на механизма на работа със споделена между два процеса памет

pic. based on `https://man7.org/linux/man-pages/man7/shm_overview.7.html`

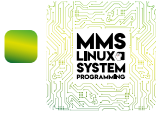
Комуникация с обща памет: предимства и недостатъци

- Положителни страни:
  - интегрална част при организацията на многопоточна работа, като породените процеси споделят памет и реферирани променливи автоматично;
  - **по-висока производителност при междупроцесна комуникация**, защото се избягва междинно копиране на буфери, а също и редица допълнителни системни извиквания;
  - **разпределя се множество от виртуални адреси**, а въвеждането и извеждането на страници от паметта е отговорност на ОС
- Недостатъци:
  - **необходима е по-комплексна синхронизация**, както програмна, така и съобразена с детайлите на микропроцесорната имплементация

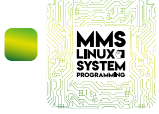


Posix shared memory: приложна секция

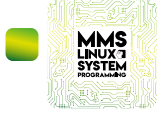
- Полезни API функции:
  - `int shm_open(const char *name, int oflag, mode_t mode);` - създава нов Posix обект, представящ регион споделена памет;
  - `int shm_unlink(const char *name);` - освобождава референцията към указания чрез аргумента име споделен обект памет;
  - `void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);` - изобразява обект, идентифициран чрез аргумента файлов дескриптор, във виртуалното адресно пространство на извикващия процес и го прави достъпен за манипулация;
  - `int munmap(void *addr, size_t length);` - освобождава референцията към споделения регион памет, указан чрез аргумента виртуален адрес



Работа с Posix споделен регион памет: разглеждане на пример:
`https://man7.org/tlpi/code/online/dist/pshm/pshm_create.c.html`

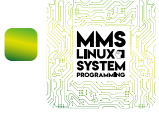


Как да преценим какъв подход и програмни инструменти да изберем?
Време за обобщение...



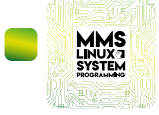
Полезни насоки и препоръки

- По отношение на жизнения цикъл на заданията избор между:
  - средства под контрола на всеки процес като обработчици на сигнали или експортирани споделени обекти в паметта;
  - примитиви, които са извън контекста на приложенията и се разглеждат като споделен комуникационен ресурс - например именувани програмни канали, изобразявани в паметта файлове и други;
  - подходи, които не налагат явно програмиране на функционалност - пренасочване на стандартен вход и изход и анонимни програмни тръби за обмен на данни



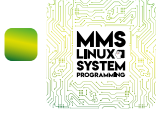
Полезни насоки и препоръки: продължение

- По отношение на изискванията за бързодействие:
  - дали е уместно да разчитаме на системния диспечер и веригите на обръщения към ядрото чрез поредици системни извиквания;
  - дали целим оптимално бързодействие и минимално разхищение на системни ресурси - тогава добър избор би бил механизмът на споделената памет;
  - дали заданието ни позволява допълнително време за програмиране на специализирана комуникационна логика (с помощта на вече дискутираните подходи) или се стремим към бърза прототипизация, при която процесите общуват неявно - само чрез пренасочване на своя стандартен вход и изход



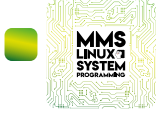
Полезни насоки и препоръки: финални такива

- Относно характера на комуникацията
  - дали алгоритъмът налага синхронизация и определен протокол на комуникацията - в този случай опашките от съобщения биха били удачни;
  - дали логиката на приложението ни е зависима от асинхроннно или синхронно постъпване на нови данни и какви времеви толеранси се допускат - отговорът се определя според сложността на решаваната задача;
  - дали алгоритъмът налага буфериране и блокиране на функционалността при липса на нови данни или не толерира това - би могло да се направи избор между четене и запис от програмни канали или арбитриране на споделени структури в паметта

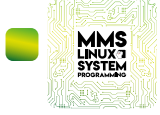


Вместо заключение...

- Някои финални бележки:
  - системното приложно програмиране създава предпоставки за креативност, като не съществуват фиксирани правила и норми;
  - изборът на механизми за междупроцесна комуникация и синхронизация се определя динамично при проектиране според изискванията на алгоритмите: за ефикасност, консумирана памет, допустимите времеви ограничения, характера на програмния товар и др.;
  - инвестирайте време и усилия в разучаването на вече достъпните инструменти и програмни техники

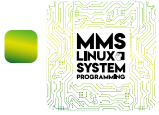


Разполагате с богат набор от налични програмни средства, които ОС и системните библиотеки ви предоставят.
Използвайте ги умело и ще подсигурите желаната функционалност.
Успешна работа!



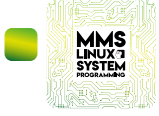
Бележки по материалите и изложението

- материалът е изготвен с образователна цел;
- съставителите не носят отговорност относно употребата и евентуални последствия;
- съставителите се стремят да използват публично достъпни източници на информация и разчитат на достоверността и статута на прилаганите или реферирани материали;
- текстът може да съдържа наименования на корпорации, продукти и/или графични изображения (изобразяващи продукти), които може да са търговска марка или предмет на авторско право - ексклузивна собственост на съотнесените лица;
- референциите могат да бъдат обект на други лицензи и лицензни ограничения;
- съставителите не претендират за пълнота, определено ниво на качество и конкретна пригодност на изложението;
- съставителите не носят отговорност и за допуснати фактологически или други неточности;
- свободни сте да създавате и разпространявате копия съгласно посочения лиценз;



Референции към полезни източници на информация

- https://en.wikipedia.org/
- https://en.wikipedia.org/wiki/Inter-process_communication
- https://opensource.com/article/20/1/inter-process-communication-linux
- https://tldp.org/LDP/tlk/ipc/ipc.html
- https://lwn.net/Articles/823513/
- https://man7.org/training/download/lusp_pshm_slides.pdf
- https://man7.org/linux/man-pages/man3/sendmsg.3p.html
- https://man7.org/conf/lca2013/IPC_Overview-LCA-2013-printable.pdf
- https://man7.org/linux/man-pages/man3/mkfifo.3.html
- https://man7.org/linux/man-pages/man2/sigreturn.2.html
- https://man7.org/tlpi/download/TLPI-52-POSIX_Message_Queues.pdf
- https://search.creativecommons.org/



Благодаря Ви за вниманието!