7. Процеси и комуникация между тях в операционната система.

Анотация: Процеси и комуникационни канали – основни абстракции, предоставени от операционната система. Неформално определение и функционални изисквания. Комуникация между процеси чрез споделена памет. Съревнование за ресурси (race condition). Хардуерна защита на ресурс, синхронизация чрез spinlock. Синхронизация от високо ниво – семафор. Приспиване и събуждане на процеси (block/wakeup). Реализация чрез семафори на комуникационна тръба (pipe), съхраняваща n елемента.

Процес – определение

Процес е инстанция на програма, която е в момент на изпълнение в дадена машина. Това са поредица от команди по време на изпълнение. Процесът може да инициира подпроцес, който се нарича детински процес (а инициращия процес се нарича негов родител). Детинският процес е частична реплика на родителския процес и споделя някои от неговите ресурси. Процесите могат да обменят информация или да синхронизират работата си чрез няколко метода за междупроцесорна комуникация (Inter process communication – IPC). Те могат да комуникират помежду си чрез: споделена памет или чрез обменяне на съобщения. Операционната система може да имплементира и двата метода на комуникация.

Комуникационни канали – основни абстракции предоставени от операционната система. Неформално определение и функционални изисквания

1. **Неименувана тръба (pipe)** се създава чрез системното извикване pipe(fd[2]). То връща два файлови дескриптора на мястото на елементите на подадения масив int fd[2]. fd[0] съхранява файловия дескриптор за четене, а fd[1] съхранява файловия дескриптор за писане. Тъй като тази тръба не е именувана, тя е видима само за процеса, който я е създал, както и от наследниците му (децата му, децата на децата му и т.н.). Тази тръба може да се използва само в системата, тоест не може да се използва в интернет. Тя служи за свръзка между процес и друг процес, който има роднинска връзка с първоначалния процес. Тъй като тръбата е неименувана, тя не използва пространството на имената.
2. **Именувана тръба (FIFO)** се създава чрез библиотечното извикване mkfifo(...). За разлика от неименуваната, този вид тръба е видима за всички процеси в системата и може да бъде ползвана от тях. Тя се използва за осъществяване на комуникация между два процеса, които не са задължително в роднинска връзка. Този вид тръба се обвързва с име, откъдето следва, че тя използва пространството на имената.
3. **Писане във файл и четене от файл**. Чрез системното извикване open(filepath, open_flags, ...) се създава файлов дескриптор, който сочи към единя край на комуникационния канал, който се изгражда в ядрото на операционната система и е за писане, а на другия край сочи към файла, който може да се разглежда като много прост абстрактен процес, служещ само за съхраняване на данни. Чрез системните команди read(int fd, void *buf, size_t cnt) и write(int fd, const void *buf, size_t cnt) може да се чете и пише от и във файла. Този вид комуникационен канал използва пространството на имената.
4. **Конекция (socket)**. Това е именуван обект, който играе ролята на крайна точка за комуникация между отдалечени обекти и който се използва за адрес при изграждането на връзка с друг процес. Тук има два вида процеси – единият се нарича сървър и работи като такъв (ще обслужва други процеси), а другият – клиент. И двата вида процеси ще трябва да изпълнят различни поредици от извиквания, за да изградят връзка помежду си, които ние няма да разглеждаме тук. Предимствата на сокетите пред тръбите е, че процесите, с които може да се изгради връзка, могат да не се намират в една и съща система. Сокета на сървъра използва пространството на имената, тъй като трябва да му бъде зададено име, с което да бъде видимо за останалите процеси. Сокета на клиента обаче не използва пространството на имената, тъй като не се обвързва с име.

Комуникация между процеси чрез споделена памет

Комуникацията между процесите, използващи споделена памет, изисква процесите да споделят някаква променлива. Използването на тази споделена променлива напълно зависи от това как програмистът ще имплементира комуникацията. Един от начините за комуникация с помощта на споделена памет може да се представи по следния начин: Да предположим че процес p1 и процес p2 се изпълняват едновременно и те споделят някои

ресурси или използват някаква информация от друг процес. p1 генерира информация за определени изчисления или използвани ресурси и я съхранява като запис в споделена памет. Когато p2 трябва да използва споделената информация, той ще провери записа, съхранен в споделената памет, и ще вземе под внимание информацията, генерирана от p1 и ще действа според нея. Процесите могат да използват споделена памет за извличане на информация като запис от друг процес, както и за предоставяне на всякаква специфична информация на други процеси. Проблемите на комуникацията чрез тръби (pipes, fifos) и обмяна на съобщенията (message passing) е, че информацията трябва да премине през ядрото, докато при комуникацията чрез споделени данни (памет) – това не е обходимо.

Съревнование за ресурси (race condition)
Race Condition на български може да се преведе като борба или съревнование за ресурси и то възниква когато два или повече процеса/нишки имат достъп до споделени ресурси и се опитват да ги променят едновременно. Алгоритъмът за даване на процесорно време за изпълнение на процес може да скача непредвидимо и по всяко време. Ние не знаем реда, в който процесите ще се опитат да получат достъп до споделените данни (този ред може да се счита за хаотичен) и следователно резултатът от промяната в данните зависи от алгоритъма за планиране на процесите. По този начин два или повече процеса се „съревновават“ за достъп до данните. Този достъп до данни, често е свързан и с някаква промяна и това може да доведе до нарушаването им, като в някой момент структурата от данни на общата памет е била нарушена временно от методите ѝ (тоест инвариантът на структурата е бил временно нарушен). Но именно ако през това време друг метод използва структурата, то ресурса ще бъде използван докато се е актуализирал – моментите, в които се актуализира се наричат критични секции. Проблемът често възниква когато един процес или нишка изпълнява операцията „провери и след това действай“ (например „провери“ ако стойността е x и след това „действай“, за да направиш нещо, което зависи от това, че стойността е била такава), а друг процес направи нещо със стойността на x между „провери“ и „действай“ на първоначално споменатия процес:

```
if (x == 5){ // „провери“
      y = x * 5 // „действай“
} /* но ако някой друг процес или нишка е променила стойността на x между
     проверката и действието, то y няма да е това което очакваме, тоест 25 */
```

(*примера по-горе е валиден и за еднопроцесорни системи*)

Въпросът е, че y може и да е 25, но може и да бъде всичко (като например парче от стринг), в зависимост от това дали друг процес е бъркал в критичен момент и в критична секция на кода. Няма как да знаем дали това се е случило. За да предотвратим появата на борба за ресурси, обикновено поставяме заключване около споделените данни, за да гарантираме, че само един процес може да има достъп до тях за определено време. Тоест нещо подобно:

```
lock(x) // заключи x
if (x == 5){
      y = x * 5 /* сега всеки друг процес или нишка няма да може да пипа x,
                   докато заключването не се отключи и следователно ще знаем,
}                  че y = 25 */
unlock(x) // отключи x
```

Без изрично/експлицитно синхронизиране, което да координира достъпа до споделени данни, нишка може да промени променливи, които друга нишка е започнала вече да използва и това може да доведе до непредвидими резултати.

Хардуерна защита на ресурс, синхронизация чрез spinlock
Повишаването на честотата на процесорите вече е станало невъзможно, поради чисто физически причини. За да се развиват системите, те стават все по-сложни с все повече ядра и повече от един процесор и тогава имаме реален паралелизъм. Паралелно работещите процесори правят метода с прекъсването неработещ. Може да прекъснем процесите от едно ядро, но процесите от друго ядро може да правят каквото си поискат и

в това число и да бъркат в критични секции. Трябва да пазим някакъв допълнителен бит в споделената структура от данни, за да индикираме чрез него за това дали тя се ползва в момента от някой друг процес или нишка. Този бит се нарича *lock*, а метода, който използваме се нарича *spinlock*.

За разсъжденията по-долу се базираме на допускането, че хардуера може да блокира прекъсвания. Този факт беше достатъчен за еднопроцесорна система. Допускаме още, че хардуера може да има специални атомарни инструкции, които да променят бит и да го прочитат след това като една единна непрекъсната инструкция.

$$
\begin{aligned}
&spin\_lock : R = test.and.set(lock)\\
&\qquad if\ (R == 1)\ goto\ spin\_lock\\
&\qquad critical\_section \begin{cases} p_1 \\ : \\ p_k \end{cases}\\
&spin\_unlock : lock = 0
\end{aligned}
$$

В момента, в който процес е в критична секция и друг процес изяви желание да я ползва, то програмата ще го връща циклично докато работещият в критичната секция процес не излезе от нея и не промени *lock bit*-а (макроса) на 0, което ще позволи на втория процес да влезе и да заключи след себе си.

В многопроцесорна система – това би трябвало да осигури защита, обаче възникват допълнителни проблеми: ами ако друга програма подбудена рекурсивно от критичната секция се извика? – *lock*-а вече ще е 1-ца и ще влезе в безкраен цикъл без да може да излезе, защото излизането и писането на 0 се случва само в края на критичната секция. Да допуснем, че по някакъв начин може да забраним на програмистите рекурсивното извикване в кртитичната секция. Но същият проблем може да се случи и при прекъсването. Нека обаче се опитаме да забраним прекъсванията:

$$
\begin{aligned}
&spin\_lock : disable\_interrupt\\
&\qquad R = test.and.set(lock)\\
&\qquad if\ (R == 1)\ goto\ spin\_lock\\
&\qquad critical\_section \begin{cases} p_1 \\ : \\ p_k \end{cases}\\
&spin\_unlock : lock = 0\\
&\qquad enable\_interrupt
\end{aligned}
$$

Нека дпуснем, че два процеса p и q искат да влязат в критичната секция и p е влязъл, а q цикли. Обаче p е забранил прекъсванията и не само p, ами и всички които чакат също са забранили прекъсванията, което е недопустимо, тъй като това е все едно да забраним на периферията да общува с всички тези циклещи процеси, което прави системата неефективна и слаба. От друга страна, другите процесори хем въртят цикли и вършат работа, хем тази работа е безполезна. Затова ще направим още едно последно подобрение и ще спрем до тук:

$$
\begin{aligned}
&spin\_lock : disable\_interrupt\\
&\qquad R = test.and.set(lock)\\
&\qquad if\ (R == 0)\ goto\ critical\_section\\
&\qquad enable\_interrupt\\
&\qquad goto\ spin\_lock\\
&\qquad critical\_section \begin{cases} p_1 \\ : \\ p_k \\ lock = 0 \end{cases}\\
&\qquad enable\_interrupt
\end{aligned}
$$

Сега вече, ако критичната секция е заета, другите процеси ще циклят, но ще имат възможността да свършат и някоя друга полезна работа през това време.

Синхронизация от високо ниво – семафор
Тъй като *spinlock*-а не реализира свойствата на хардуера, които ги имаме – като прекъсвания (а и самият той е непрекъсваем), както и за да може да реализираме пъргави системи за управление, е необходимо да може да осигурим възможност на критичните секции да са прекъсваеми и да има смяна на контекста. Това става с похвати на по-високо ниво, като един от тях е семафорът, който е въведен от Едгар Дейкстра през 60-те години на 20-ти век (приблизително през 1965 г.).

Семафорът е абстрактен механизъм за синхронизация, който предпазва ресурс от прекомерно използване, за да не настъпи катастрофа. Най-грубо казано, той представлява ключалка с брояч, който казва колко процеси/нишки могат да се намират едновременно в критичната секция, която пази ключалката. Структурата на данните му е следната:

Два атрибута:

- $cnt : int \rightarrow$ брояч (показва колко процеса могат да преминат бариерата, за да ползват споделения ресурс). Ако $cnt < 0$, то $|cnt|$ е броя на приспаните процеси;
- $L : list \rightarrow$ контейнер/списък, който може да го разглеждаме като множество от приспани процеси.

Силен семафор е този, който използва обикновена опашка за множеството L. Слаб семафор е този, при който има някакъв критерии, който се ползва за приоритет при обслужването на процес. Слабостта идва от факта, че такъв критерии може да причини така нареченото „гладуване“ на процеси/нишки с най-нисък приоритет, което се счита за проблем със синхронизацията. Не искаме да имаме процеси, които да чакат твърде много.

три метода:

- $init(int\ cnt = 0) \rightarrow$ инициализатор/конструктор на семафор, който приема аргумент по подразбиране – цяло число, с което се инициализира брояча;
- $wait() \rightarrow$ с този метод възлагаме работа на друг процес или искаме да ползваме общ ресурс. Опитва се да достъпи общ ресурс, ако получи достъп – декрементира брояча с 1, ако не получи – влиза в списъка с приспаните процеси;
- $signal() \rightarrow$ освобождава общия ресурс (аналог на хардуерно прекъсване).

Хубавата операционна система предоставя готови комуникационни канали и възможност за ползване на семафори като допълнителен механизъм. Реалното място, където масово се използват семафорите е в ядрото на операционната система.

Приспиване и събуждане на процеси (block/wakeup)
Процесите се характеризират с някакво текущо състояние в което се намират. Едно от тези състояния е sleeping/blocked.

Спящите процеси от гледна точка на потребителя са работещи програми, които са стигнали до състояние, при което нямат нужда да изчисляват нещо докато не настъпят интересни за тях събития в системата. Те са в комуникация с друг процес или устройство и очакват да им се подадат данни, но очакваните процеси нямат готовност да им подадат (например поради запушен комуникационен канал или поради бавна работа на другата страна и т.н.). От гледна точка на реализацията може да чакат:
- I/O – очаква извършване на входно изходни операции;
- Time – очаква да настъпи времеви момент;

- Signal – очаква сигнал от друг процес за промяна на състоянието му (на другия процес);
- Процеса бива приспан, защото страницата, с която иска да работи не е на реалната памет, а е някъде на твърдия диск;
- Чака да се освободи семафор от предишен ползвател.

Процес не може произволно да реши да се приспи. Трябва да има някаква причина за това.

За да се събуди спящ процес, ядрото изпълнява алгоритъм за събуждане. Алгоритъмът освобождава дадена ключалка и събужда всички процеси, които чакат тази ключалка да се освободи. Аналогично процес може да се събуди, ако получи сигнал, че дадена входно-изходна операция е приключила. Ядрото повишава нивото на изпълнение на процесора при събуждане, за да блокира прекъсванията. След това за всеки процес, който е заспал чакайки събитие което вече е настъпило бива вкарван от състояние на спящ в състояние на готов за изпълнение (sleeping → ready to run). Премахва процеса от свързания списък със спящи процеси, поставя го в свързан списък с процеси, отговарящи на условията за планиране. Събуждането не води до незабавно насрочване на процес. Това само прави процеса допустим за планиране.

Реализация чрез семафори на комуникационна тръба (pipe), съхраняваща n елемента

Предполагаме, че тръбата може да съхранява до n байта, подредени в обикновена опашка. Тръбата се използва от няколко паралелно работеши „изпращачи/получатели“ на байтове. Процесите изпращачи – слагат байтове в края на опашката, а получателите – четат байтове от началото на опашката.

За да реализираме комуникационната тръба ще използваме следните инструменти:

- Опашка (или масив) Q с n елемента – тъй като е възможно да имаме бърз и бавен процес (например: единият чете бързо, а другият пише бавно) е нужно опашката да бъде по-дълга, за да не се приспива по-бързия процес (приспиването е бавна операция, тъй като включва в себе си смяна на контекста). В началото тази опашка ще е празна;
- $free\_bytes$ – семафор, който ще индикира за свободните байтове в опашката Q;
- $ready\_bytes$ – семафор, който ще индикира колко байта са готови за четене (до колко е запълнена опашката Q);
- $mutex\_read$ – мутекс, който ще защитава критичната секция за четене;
- $mutex\_write$ – мутекс, който ще защитава критичната секция за писане.

Инициализация:

$free\_bytes.init(n)$
$ready\_bytes.init(0)$

$mutex\_read.init(1)$
$mutex\_write.init(1)$

$\underline{READ}$	$\underline{WRITE}$
$byte\ b$	$byte\ b$
p_1	q_1
p_2	q_2
:	:
$ready\_bytes.wait()$	$free\_bytes.wait()$
$mutex\_read.wait()$	$mutex\_write.wait()$
$b = Q.get()$	$Q.put(b)$
$mutex\_read.signal()$	$mutex\_write.signal()$
$free\_bytes.signal()$	$ready\_bytes.signal()$
:	:

Процесът, който чете, първоначално ще проверява дали има готови за четене байтове. Ако няма, той ще се приспи и ще чака да постъпят такива. Ако има, той ще провери дали някое друго копие на P не чете в момента. Ако да, той отново се приспива. Ако не, той ще прочете байт и ще го запише в променливата b. След това ще сигнализира, че в опашката има още един свободен байт чрез подаване на сигнал на семафора $free\_bytes$.

Процесът, който пише, първоначално ще провери дали има свободни байтове в опашката. Ако няма, ще се приспи и ще чака да се освободят байтове. Ако има, ще трябва да провери дали някое друго копие на пишещ процес не пише в момента в опашката. Ако да, то отново процеса ще се приспи. Ако не, ще сложи байта си b в опашката и ще сигнализира че още един байт е готов за четене, което се осъществява чрез подаване на сигнал на семафора $ready\_bytes$.

Кодовете по-горе, показващи как може да се реализира комуникационна тръба чрез семафори са валидни само за синхронни входно-изходни операции, тъй като процесите се приспиват, когато има недостиг на байтове и по този начин се изчаква за да може да се окомплектоват.