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

1.**Процеси и комуникационни канали** - Процесът е програма в хода на нейното изпълнение, абстрактна машина за обработка на информация. Процесът е реален изчислителен процес, който обработва информацията в реалната среда, той не съществува изолирано, самостоятелно, а си взаимодейства с други обекти в информационната среда – с други процеси и файлове. Когато процесите си комуникират, те излизат от локалното си пространство, за да използват споделените ресурси предоставяни от операционната система - достъп до споделена памет, общи регистри или файлови дескриптори.

Повечето от функционалностите, които една ОС поддържа са удобства за програмистите. Езиците за програмиране са абстрактни, не се влияят от хардуера, това са езици от високо ниво (с++, python и др.). Веднъж написана програмата на такъв език, тя може да бъде превърната в процес като част от ОС и да съществува дълго време без корекции.

Идеята на ОС е да можем да поддържаме такава дълготрайност на програмисткия труд, защото хардуерите се променят често и бързо. Основната цел на процеса е да поддържа тази вечност и абстрактност.

Относно комуникационните канали - ако представим процесите като отделни изчислителни инструменти, които комуникират чрез прости методи за връзка, ние дефинираме процес и начина му на взаимодействие с останалите, ОС ще поеме действието по реализацията на комуникационните канали. Например при писане на програма и изискване от нас да отворим файл, казваме на ОС да намери файл Х и да го отвори и по този начин ОС ни предоставя първия свободен файлов дескриптор, едно неотрицателно число (номера на комуникационния канал, с който взаимодействаме, уникален идентификатор на отворения файл), след което ние пращаме или получаваме байт след байт информацията по предоставения ком. канал.

Освен изискванията за това процесът да си комуникира с други обекти и да е абстракция за работеща програма, която да решава конкретна изчислителна задача, трябва да поддържаме и илюзията за много едновременно работещи процеси. За да може за потребителят да се поддържа илюзията, че всички процеси работят едновременно, техните изпълнения се делят на малки интервали; за превключване на процесите, съществува хардуерен елемент (таймер, вътрешен часовник – прекъсване, което е част от интегралната схема), който преценя дали даден процес работи и трябва ли да се прекрати изпълнението му ако си е изразходил малкия интервал процесорно време.

Task scheduler е структурата която решава това кога да се превключва процеса (от чакащ в работещ) – осъществява се чрез хардуерен елемент, който на определен интервал пуска съобщение timer interupt. Time-sharing се нарича бързото и често превключване на процеси. Съответно всеки процес редува процесорно време, с период на изчакване. Има два вида изчакване:

-blocking - когато чака друг процес или устройство да му предостави данни и тогава да продължи да работи

-time-interrupt прекъсване, когато таймера го изкл, защото е бил активен много дълго време и изчаква да му бъде предоставен нов квант за смятане

И двете форми на смяна на управление водят до случаен елемент, т.е не може да бъде предсказано кога даден процес ще бъде прекъснат и как ще се подредят инструкциите във времето.

**2. Споделена памет, race condition**

Заради невъзможността да се предвиди работата на процесите във времето и от друга страна нуждата от използване на споделени ресурси е необходимо да се установи начин за синхронизацията на "паралелно" работещите процеси и начина по който те достъпват общи ресурси. В противен случай може да възникне съревнование за ресурси.

Race condition (съревнование за ресурси) е проблем, при който няколко процеса използват общ ресурс. Всеки процес, който работи със споделен ресурс има поне едно място в кода си, в което той използва този споделен ресурс. Тази част от кода се нарича критична секция (critical region). Желанието да се синхронизират процесите използващи споделения ресурс Х се свежда до това да се направи така, че винаги само един процес да е в своя критична секция , докато всички други изчакват излизането му от този регион, т.е никой друг процес не може да е в критичната си секция използвайки същия споделен ресурс.

Съществуват няколко начина за осъществяване на това, но по същество те се делят на методи за синхронизация от ниско и високо ниво. Еволюционно първият пример за справяне с този проблем е забраната на прекъсванията докато процес е в своята критична област. Първият процес, който влезе в своя критичен регион няма да може да бъде прекъснат докато критичната секция не завърши. Това обаче е приложимо само за еднопроцесорна ОС.

За многопроцесорни ОС може да се използва подход наричан spinlock. При него се създава променлива turn. Тя приема стойности 0, когато ресурса е свободен и 1, когато е зает и показва дали процес може да влезе в критичната си секция или трябва да изчака. Когато процес0 влиза в своя критичен регион чрез специална команда той променя стойността на turn на 1 и така сигнализира, че използва споделеният ресурс контролиран от тази променлива. През това време процес1 изчаква в цикъл, постоянно проверявайки за промяна на променливата (busy waiting). Тъй като нишката остава активна, но не извърша смислена задача, то се казва че тя извършва активно чакане. Минус на подхода е, че ако критичната секция е дълга,то за много използвано процесорно време ще е свършена нулева работа. Освен това трябва допълнително да се гарантира (чрез атомарната инструкция tsl), че няма да се получи race condition за промяната и четенето на тази turn променлива. Плюс е, че това е бърз механизъм при кратки критични секции и няма друг механизъм от ниско ниво.

3. **Синхронизация от високо ниво** **- семафори**

Съществуват и структури от високо ниво, които служат за синхронизация. Те обикновено се основават на принципа за приспиване и събуждане на процес. Ако един процес няма, работа, която да извърши в даден момент и очаква външно събитие, то той бива приспан до случването на събитието. Когато споделеният ресурс е наличен отново, процесът се събужда и продължава изпълнението си от там докъдето е стигнал. Най-популярната реализация на тази абстрактна идея е семафорът.

Семафорите са структура представена от Дийкстра за синхронизацията на процеси, използващи общи ресурси. Имплементацията на семафор изисква съставянето на структура поддържаща прост интерфейс от няколко метода.

Структура:

-counter - число. То не може да се чете, а само да се увеличава и намалява

-списък на приспаните процеси

Методи:

init() - функция за задаване на начална стойност на брояча на семафора, когато той се инициализира

wait() - функция, която намалява брояча с 1; ако брояча <= 0 нишката се блокира и чака, той да стане положително число.

signal() - функция, която увеличава брояча с 1 и така сигнализира, че е освободила ресурс и е приключена критична секция. При увеличаване на семафора, една от чакащите нишки бива отблокирана и допусната в своята критична секция.

Трябва да се отбележи, че променливата за брояча на семафора, е недостъпна за процеса, който използва семафора, защото не е сигурно че след четенето, тя няма да се промени от друг процес. Ето защо при извикване на wait() не може да се гарантира, че процесът няма да бъде блокиран. Също така не може да се предвиди дали ще има освободени нишки след извикване на signal(). След изпълнението на signal() събуденият процес продължава изпълнението си заедно със този, който е направил signal. (concurrently)

Решение на consumer-producer със семафори според LBS

Общ код:

|  |
| --- |
| mutex\_P = Semaphore(1)  mutex\_C = Semaphore(1)  items = Semaphore(0)  spaces = Semaphore(n)  local event - локална променлива за всяка различна нишка |

|  |  |
| --- | --- |
| **consumer code**  **items.wait()**  **mutex\_C.wait ()**  **event = buffer.get ()**  **mutex\_C.signal()**  **spaces.signal()**  **event.process()** | **producer code**  **event = waitForEvent ()**  **spaces.wait()**  **mutex\_P.wait()**  **buffer.add ( event )**  **mutex\_P. signal ()**  **items.signal ()** |