1.

Структурите данни, необходими за реализацията на семафор са:

1. Брояч cnt, който пази броя на процесите, които имат достът до ресурса, охраняван от семафора
2. Контейнер Q, който пази информация за процесите, които чакат достъп до ресурса

Процедурите, необходими за реализацията на семафор са:

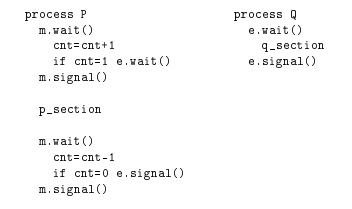
1. Конструктор init(int), който дава начална стойност на брояча cnt и инициализира контейнера Q да е празен.
2. Метода wait(), който се ползва при опит за достъп до ресурс (заемане на ресурс). Броячът cnt се намалява с 1. Ако той стане отрицателен, процесът, извикващ wait(), се блокира и номерът му се вкарва в контейнера Q.
3. Метода signal(), който се ползва при завършване на достъпа до ресурс (освобождаване на ресурс). Броячът cnt се увеличава с 1. Ако контейнерът Q не е празен, се изважда процес от него и се активира.

Семафорът е силен, ако за реализацията на контейнера Q се използва обикновена опашка, т.е. процесът, който е приспан най-рано, първи ще се събуди. Семафорът с слаб, когато за реализацията на контейнера Q не се използва обикновена опашка. Тогава при извикването на signal() може да се събуди процес, който не е първи в списъка на чакащите, т.е. има приоритизация.

Опишете максимално несправедлива ситуация, която може да се случи в избирателната секция, ако на входа на секцията пазач – член на избирателната комисия, пуска гласоподавателите така: 1 – във всеки един момент в секцията може да има най-много двама гласоподаватели, 2 – пазачът работи като слаб семафор:

Ако пазачът на входа на избирателната секция действа като слаб семафор, може да се случва следната неприятна ситуация: първите двама гласоподаватели влизат в секцията, пристига трети гласоподавател (неприятел на пазача) и чака. След него започват да пристигат други приятели на пазача и той ги пуска с предимство. Може да се стигне дотам, че третият гласоподавател чака цял ден и гласува последен. Това се нарича starvation (гладуване).

2.



Първоначално забелязваме, че семафорът m се използва само от процеса P. Той е мютекс и е пазач на брояча cnt. В неговата опашка може да има само копия на процеса P. Eдно работещо копие на процеса P може да увеличава/намалява cnt, което е свързано със заемането/освобождаването на семафор e. Заемането се случва преди критичната секция на P, а освобождаването след нея. Лесно се вижда, че когато не се изпълнява никакво копие на Q, може да сe изпълнява критичната секция на P произволен брой път. Броят на изпълняването на критичната секция е равен на стойността на брояча cnt => 2) Да. Заемането на семафора e става точно когато cnt се променя от 0 на 1, а освобождаването става точно когато cnt се променя от 1 на 0. Заемането и освобождаването на семафор е и в двата процеса е в един и същи ред:

* Ако първо се изпълни някакво копие на процес Q, то ще премине през е.wait(), защото при инициализирането виждаме, че броячът е 1 и ще изпълни критичната си секция, но други копия не могат, защото броячът ще е станал 0 => 3) Не.
* Ако първо се изпълни някакво копие на процеса P, то ще изпълни критичната си секция и също може и други копия да я изпълняват. Това ще е докато cnt>0. Когато cnt стане 0 Q ще може да изпълни критичната си секция, но P няма да може. Следователно няма как да се изпълняват двете критични секции едновременно => 1) Не.

Няма условие за deadlock. Процесът Q не може да инициира deadlock, тъй като ползва само един ресурс. Процесът P също не може да инициира deadlock, заради подредбата на заемане на ресурсите (първо семафор m, после семафор e). Условие за гладуване може да се появи при процес Q. Когато P започне изпълнението си и нови копия започнат работа по критичната секция, то cnt може да е положителен за неограничено време, при което и Q ще бъде приспан за неограничено време.

3.

При синхронна входно-изходна операция, системното извикване може да блокира потребителския процес, поръчал операцията. Същевременно, при нормално завършване, потребителският процес очаква коректно комплектоване на операцията – четене/запис на всички предоставени/поръчани данни във/от входно-изходния канал, или цялостно изпълнение на друг вид операция.

При асинхрoнна входно-изходна операция, системното извикване не може да блокира потребителския процес, поръчал операцията. Същевременно, ако операцията не се комплектова, ядрото връща управлението на процеса със специфичен код за грешка и друга информация, която служи за определяне нивото на завършеност на операцията. Потребителският процес трябва да анализира ситуацията и при нужда да направи ново системно извикване по-късно, с цел да се довърши операцията.

Използването на асинхронни операции позволява на един процес да извършва паралелна комуникация по няколко канала с различни устройства или процеси, без да бъде блокиран в случай на липса на входни данни или препълнен буфер с изходни данни, или друга ситуация, водеща до блокиране.

Първи пример: Когато ползваме browser, той трябва да реагира на входни данни от мишката и клавиатурата, както и на данни, постъпващи от интернет, т.е. на поне 3 входни канала. Браузърът проверява чрез асинхронни опити по кой от каналите постъпва информация и реагира адекватно.

Втори пример: Сървър в интернет може да има много на брой клиентски програми, като поддържа отворени TCP връзки към всяка от тях. За да обслужва правилно клиентите, сървърът трябва да използва асинхронни операции, за да следи по кои връзки протича информация и кои са пасивни.

4.

В съвременната ОС същестуват няколко процесни състояния: running (използва CPU), ready (чака CPU), sleeping (чака вход/изход или чака време), stopped – спрян, zombie – започнал е изпълнението си, но не го е завършил.

4

3

2

1

Преход 1 се случва, когато един процес открие, че не може да продължи изпълнението си (чака вход/изход или чака време). Преход 2 и 3 се управляват от модул на ОС, който се нарича scheduler, без процесът да го иска. Когато планировчикът прецени, че един процес достатъчно дълго време е използвал ЦП, той насилствено му го отнема и го предоставя на друг ready процес (преход 2). Koгато ЦП се освободи поради блокиране, завършване или сваляне на текущ процес, планировчикът избира един от ready процесите и го прави running (преход 3). Преход 4 се осъществява, когато процесорът свърши работа с ресурс и го предоставя на други процеси да го използват.

5.

Ситуация, при която два или повече процеса четат и пишат в обща памет и крайният резултат зависи от реда, в който работят процесите, се нарича race condition. Ако прекъсванията са непредсказуеми или се изпълняват програми паралелно, се получава разрушаване на информация. За да се предотврати това, конкурентните процеси трябва да се синхронизират.

Възможно ли е да настъпи race condition при еднопроцесорни системи? Да, възможно е, когато са пуснати няколко нишки. Предотвратяването става чрез забрана на прекъсванията. То гарантира взаимно изключване, тъй като това забранява превключването на контекста на процесора и прави невъзможно преплитането на процеси.

Какви инструменти използваме, за да избегнем race condition? Проблемът за избягване на състезанието е бил формулиран от Дийкстра чрез термина критичен участък. Част от кода на процеса реализира вътрешни изчисления, които не могат да доведат до race condition. В друга част на кода процесът осъществява достъп до обща памет или върши неща, които да доведат до race condition. Трябва да се изгради механизъм, чрез който да се гарантира, че когато един процес е в критичната си секция, никой друг процес не може да влезе в своята критична сеция. Този механизъм трябва да притежава свойствата:

* Взаимно изключване или mutex (mutual exclusion) – във всеки един момент от време само един процес може да се намира в критичната секция
* Процес, който се намира в критична секция не може да влияе на другите процеси
* Не трябва да се забавя влизането на процес в критична секция, ако в момента няма друг процес в критична секция.

Всяка критична секция трябва да започва със spinlock (забраняваме на другите процеси да достъпват секцията) и да завършва със spin-unlock (позволяваме на другите процеси да достъпват секцията).

6.

P

p1

s.wait()

p2

s.signal()

p3

За да осигурим чрез семафор синхронизация, така че най-много едно копие на процеса да изпълнява p2 във всеки един момент, инициализираме sem s; s.init(1) и модифицираме кода.

Не е възможно да настъпи deadlock, тъй като копията на P заемат само 1 семафор. Starvation може да настъпи само ако семафорът е слаб. Ако всяко копие на P има различен приоритет и постоянно идват копия с висок приоритет, копията с по-нисък приоритет могат да чакат неограничено (да гладуват).

7.

Процесът P създава тръба pipe с извикване на функцията pipe(int pipefd[2]). Тръбата е достъпна само чрез файловите дескриптори pipefd[0] и pipefd[1]. Te са видими само за процеса P и неговите наследници. Процесите, които не са наследници на P не могат да ползват тръбата.

За да бъде използван от произволен процес в изчислителната среда, комуникационния канал трябва да бъде видим (именуван). В повечето UNIX системи има възможност за създаване на именувана тръба FIFO, но тя обаче се създава от един процес и е видима за всички останали, а ние искаме да е достъпен само до процеса, който го е създал.

Друг вариант е един процес да създаде именуван обект, който да послужи като адрес при изграждането на връзка от друг процес. Използваната абстракция се нарича socket. Сокетът се дефинира като единия край на комуникационния канал.

Един процес, наричан обичайно сървър, изпълнява следната поредица:

1. Създава socket:

sfd = socket(domain, type, protocol);

1. Присвоява има на сокета:

bind(sfd, &my\_addr, addrlen);

1. Започва приемане на заявки за връзки:

listen(sfd, backlog);

1. Приема заявка за играждане на връзка:

cfd = accept(sfd, &peer\_addr, addrlen);

Друг процес, наричан обичайно клиент, изпълнява следната поредица:

1. Създава socket:

fd = socket(domain, type, protocol);

1. Подава заявка за изграждане на връзка:

connect(fd, &server\_addr, addrlen);

Сървърът създава сокета sfd и му дава име чрез bind. Извикването listen активира процеса на създаване на връзки. Клиентът създава сокета fd. Извикването connect е заявка за изграждане на връзка към именувания сокет sfd. Сървърът приема заявката чрез accept. Изградена e връзка между sfd и fd (обмен на информация). Файловият дескриптор sfd продължава да приема нови заявки от клиенти. Благодарение на присвоеното му име, sfd дава възможност на другите процеси да се свържат със сървъра. Името определя какви клиенти могат да ползват сървъра. Ако то е име в интернет (IP, адрес, порт), всички процеси, изпълнявани на компютри, имащи достъп до интернет, могат да се свържат към сървъра.

8.

Задачата за читателите и писателите е модел на достъп до обща база данни от много конкурентни процеси, които се делят на 2 вида – читатели, които само четат данните, и писатели, които по някакъв начин изменят данните в базата. Искаме във всеки момент :

1. Никой да не достъпва защитения ресурс;
2. Само 1 писател да достъпва общия ресурс;
3. Няколко на брой читеталя да достъпват общия ресурс.

Ако имаше само един читател и само един писател, то решението ще е mutex:

sem room\_empty

room\_empty.init(1)

Readers:

room\_empty.wait()

Четене

room\_empty.signal()

Writers:

room\_empty.wait()

Писане

room\_empty.signal()

Решение с няколко читатели:

int r = 0 – брояч на читателите

sem room\_empty, m, t

room\_empty.init(1)

m.init(1) – mutex, за да се избегне race condition

t.init(1) - добавен по-късно, за да се избегне гладуване

Readers:

t.wait()

t.signal()

m.wait()

r = r +1

if (r == 1) then room\_empty.wait()

m.signal()

четене

m.wait()

r = r – 1

if (r == 0) then room\_empty.signal()

m.signal()

Writers:

t.wait()

room\_empty.wait()

Писане

t.signal()

Room\_empty.signal()

Aко никой от процесите не осъществава достъп до базата данни, то двата семафора са с брояч 1 и r e 0. Тогава първият процес, който се появи, ще изпълни room\_empty.wait() и ще затвори този семафор(броячът му ще стане 0).

1. Ако това е читател, то r ще стане 1 и следващите читатели няма да проверяват семафора room\_empty, а само ще увеличават r и ще започват да четат от базата данни. Когато читателят завърши четенето, той намалява брояча. Последният читател изпълнява room\_empty.signal() и ще отвори семафорът room\_empty(броячът му ще стане 1) и ще се събуди писател.
2. Ако първият процес, получил достъп до базата данни е писател, то първият читател ще бъде блокиран по семафора room\_empty, а следващите читатели ще бъдат блокирани по семафора m. Следващите писатели също ще бъдат блокирани по семафора room\_empty. Когато писателят приключи с писането, той ще изпълни room\_empty.signal() и ще събуди първият читател, той ще изпълни m.signal() и ще събуди следващия читател, който пък ще събуди следващия читател и т.н., докато всички читатели не бъдат събудени.

9.

#include <unistd.h>

#include <stdio.h>

int main(void){

int p1, p2;

p1=fork();

p2=fork();

printf(“Hello, world!\n”);

}

„Hello, world!\n“ ще се отпечата 4 пъти. 2^n е крайният брой процеси, където n е броят пъти на извикване на fork(). В нашия случай има извикване на fork() 2 пъти => 2^2 = 4 процеса, които ще изпълнят printf по веднъж.

fork() прави копие на работещия процес. Новият процес се нарича child. Процесът, който създава child се нарича parent. И двата процеса протичат едновременно, но в различна памет.

Child процеса е дупликат на parent процеса с няколко разлики:

1. Child-a има свое уникално process ID
2. Parent process ID-то на child-a е същото като process ID-то на parent-a

Тъй като са почти идиентични процесите, ние също можем да ги различим по това, което връщат. fork() връща 0 при детето и >0 при бащата (това, което е >0 е pid-а на новосъздадения процес).

В нашата програма ще се изпълни fork(), при което вече ще работят два процеса – parent и child. След това се изпълнява fork() и при parent, и при child, при което и parent, и child се дублицират и вече работят 4 процеса.

10.

Основните комуникационни канали в Linux са pipe, връзка между процес и файл и сокет (връзка процес-процес).

Неименувата тръба pipe:

Функцията pipe() приема един аргумент – масив от два елемента - int pipefd[2]. Тръбата е достъпна само чрез файловите дескриптори pipefd[0] и pipefd[1]. Te са видими само за процеса, създал тръбата, и неговите наследници. Първото цяло число в масива е за четене, докато второто цяло число е за запис.

Връзка процес-файл:

Командата за играждане на комукационен канал с файл е open. Първият аргумент на командата е char \*pathname – абсолютен или относителен път. Вторият аргумент е flag (integer). Флаговете задават свойства на комуникационния канал. Някои свойства се отнасят за това дали се чете/пише/и двете във/от файла. Ако е включено свойството O\_CREAT, тогава се създава файлът, ако го няма; O\_TRUNC – дали да отреже от файла (от там където е указателя нататък) и други. Операцията за затваряне на комуникационния канал е close(fd) , т.е. прекратява връзката между процеса и файла и освобождава файловия дескриптор fd. Open и close връщат 0 при успех и -1 при грешка. Операциите за четене и писане са read и write. Аргументите на read са файлов дескриптор (комуникационен канал, който е подготвен за работа), указател към масив от найтове (buf) и колко байта искаме да прочетем (count). Към write са същите. Резултатът, който връщат read и write e колко байта реално са прочетени.

Двустранна връзка процес-процес (connection) – socket.

Всички освен обикновената тръба използват пространството от имената.

11.

Изисквания, които удовлетворява съвременната файлова система, реализирана върху блоково устройство:

Устройствата, върху които са се реализирали файловите системи са били твърди дискове и в резултат на това реализациите са зависели от физическите особености на тези дискове. Файловете трябва да се разполагат върху диска в сектори и всеки файл трябва да има метаданни за това на кой сектор се разполага. В миналото се е правило, така че да няма много фрагментация (разхвърляне на файловете върху диска) между отделните сектори, но в съвременните операционни системи има твърде много процеси и понятията фрагментация и дефрагментация нямат смисъл. Също така достъпа до диска трябва да е бърз (lift algorithm) и трябва да е надеждно съхранението на постоянните файлове (journal).

Отложен запис (delayed write):

Отложен запис е подход, при който данните се изменят в системния буфер, но при завършване на write, физическото записване на диска може да не е завършило.

Алгоритъм на асансьора:

Нека имаме няколко процеса, които работят с файлове. Всеки от тях отправя заявки за работа с диска. Ако обработваме тези заявки в проста структура, например опашка, разпределението на секторите, които трябва да се обработят, ще бъде хаотично и изпълняването на заявките в реда, в който постъпват ще е бавно, защото главата трябва да прави много разходки между далечните сектори. Можем да оптимизираме този процес, ако заявките не изпълняваме в реда, в който са постъпили, а ги разместим във времето, така че да не прави много движения. Тази техника се нарича lift algorithm. Реалният алгоритъм работи малко по-сложно. Нека си представим, че работят паралелно потрелски процеси и алгоритъма, който пише/чете на/от диска. Всички процеси и процеса на този алгоритъм работят споделено с диска. Ако си представим диска като една отсечка, главата ще има една позиция C и наляво и надясно от нея ще има сектори, които са заявени за четене. Най-подходящата структура за тези процеси за 2 приоритетни опашки – от тази отляво ще вадим най-десния елемент, а тази отдясно ще вадим най-левия елемент. Когато процес пуска нова завка, ако заявката е наляво от С, той ще я пусне в левия heap, а иначе – в десния. Когато дисковия контролер е готов, той ще провери дали в посоката му на движение има елементи в опашката. В противен случай – ще смени посоката на движение. Тъй като тези хийпове са споделен ресурс, достъпът до тях трябва да става посредством общ mutex за всички процеси и процесът на алгоритъма.

Поддържане на журнал на файловата система:

Разместването на заявките поражда друг проблем – може при внезапно спиране на системата да се загуби информацията за необслужените заявки и така ще се повредят структурите на файловата система. За да се предотвратят такива ситуации, се използват други техники, които се използват най-напред в базите данни. Таблиците в БД трябва да се поддържат на физически носител и ако по време на транзакцията се случи нещо, тогава ще се наруши физическата репрезентация на данните. Когато започваме транзакцията, записваме действията по транзакцията като поредица от команди. Транзакциите не се извършват с реалното представяне на данните върху постоянния носител, а ги записваме в друг файл – журнал. Там се извършва транзакцията. Когато журнала се препълни, спираме транзакциите и новото състояние БД се актуализира като се взимат предвид изпълнените транзакции и се освобождава място в журнала. При внезапно спиране на системата, журнала ще пази направените промени и ще сме в старото състояние. Ако системата спре внезапно по време на транзакция, указателя в журнала ще сочи, че сегашната транзакция все още не се е изпълнила, т.е. първо се четат страите данни и тогава ще се изпълни транзакцията наново. Журналът се намира на различни места в зависимост от различните версии на ОС. Един вариант е да се слага върху диск, или върху друг дисков дял.

12.

Времето за достъп до секторите зависи от текущото положение на механичните компоненти на диска – над коя пътечка е главата и каква е позицията и над пътечката?

Твърдият диск има следните специфични физически характеристики: представлява метален диск, който се върти около своята ос. Информацията на него е записана в концентрични окръжности, които се наричат пътеки (или цилиндри). Всъщност, пътеката не е концентрична окръжност, а представлява ивица, която е оградена от 2 такива концентрични окръжности. Върху металната повърхност има нанесен магнитен слой и пътеката е разделена на голям брой битове. Този бит се съхранява на принципа на намагнитване (ако е намагнитен в една посока се брои за 1, а в друга посока - за 0). Над диска се намира устройство(глава), което засича магнитното поле на материала, който се върти под него(и разбира дали текущият бит е 0 или 1). Самата глава е закрепена на някакво рамо (подобно на грамофонна плоча). Рамото може да се премества по различните пътеки.

13.

Реализация на комуникационна тръба чрез семафор. Предполагаме, че тръбата може да съхранява до n байта, подредени в обикновена опашка. Тръбата се ползва от няколко изпращачи/получатели на байтове. Процесите изпращачи слагат байтове в края на опашката, получателите четат байтове от началото на опашката. Този метод е познат още като „producers-consumers“.

Нека имаме процесите А и B:

A

a1

sB.wait()

sA.signal()

a2

B

b1

sA.wait()

sB.signal()

b2

Нека инструкциите да се изпълняват в следния ред: a1 < b2, b1 < a2

Инициализираме семафорите sA и sB:

sA.init(1);

sB.init(0);

Разрешаваме на двата процеса да работят паралелно, както и да има цикли в тях. Ако тръгнем да изпълняваме първо процеса А, то sB.wait() ще го приспи. Обаче, другият процес ще може да се изпълни, защото броячът на sA e 1. Докато B е в дадената секция и двата семафора ще са с брояч 0. Когато завърши кодът семафорът sB ще се отблокира. Сега, ако отново се изпълни B, той ще заспи, но А е свободен да изпълнява своята секция. Виждаме, че процесите от тип А и тип В ще се редуват и дедлок няма да настъпи. Точно това е идеята, която ни позволява да конструираме комуникационна тръба. Ако си представим, че процесите от тип B пишат в тръбата, а процесите от тип А са консуматори, благодарение на този механизъм няма да има 2 поредни заявки за четене, нито 2 поредни заявки за писане.

14.

Задачата за философите – пример за дедлок:

5 философа са професори. Всеки си има специални възгледи и всеки ден обядват в столовата на университета. Сядат на маса, на която има 5 чинии и 5 вилици. Когато им сипят спагети, всеки философ иска да яде с 2 вилици. Ако всички следват общ алгоритъм(примерно всеки хваща лявата и после дясната от себе си вилица) и сядат едновременно, ще изпаднат в deadlock, защото всеки ще вземе лявата от себе си вилица и няма да има дясна, защото вече ще е взета. Тази задача показва deadlock в термина на процеси и ресурси, а не в рамките на семафори.

За да не възникне deadlock трябва да са удовлетворени следните 4 правила:

1. Всеки ресурс може да бъде използван едновременно само от 1 процес (взаимно изключване)
2. Очакване на допълнителен ресурс
3. Ползването на ресурса от даден процес не може да бъде прекъснат от външен фактор (процесът сам трябва да го освободи)
4. Цикличност

Цикличност - ако означим философите с a,b,c,d,e и вилиците си 1,2,3,4,5. Нека сложим процесите и ресурсите в граф с върхове a,b,c,d,e,1,2,3,4,5, а чрез ребра в графа са означени нуждите на конкретен философ от конкретна вилица. Да предположим, че ребрата са (a,1) (a,2) (b,2) (b,3) (c3,c4) (d4,d5) , (e5,e1) , т.е. всеки иска да вземе съседните си. Винаги когато представим ситуация на deadlock по такъв начин с граф, графът е цикличен. Tази цикличност може да се използва, за да се предотврати зацикляне. Има различни стратегии за избягване на deadlock, но няма единно правило. Ако номерираме ресурсите и процесите се опитват да достъпват ресурсите с най-малък номер, така няма да настъпи deadlock.

Другите 3 правила също могат да се използват за избягване на deadlock.

1. Това, че всеки ресурс може да бъде използван едновременно от 1 процес може да бъде подсигурено да не води deadlock, например, ако защитим ресурса със специален процес, който да поеме ролята на употреба на ресурса в режим на взаимно изключване, а останалите процеси, които искат да използват ресурса, да използват този процес. Типичен пример за това е употребата на периферни устройства, които са уникални за системата – принтер/мрежова карта. Когато имаме принтер, свързан към системата, има процес, отговорен за печатането. Процесите, които искат да използват принтера се обръщат към тази програма (обикновено се нарича **spooler**) , а не поемат контрола над принтера сами.

2, 3- За да не ги нарушим, понякога се налага да ползваме по–фино методите за синхронизация м/у процесите. Примерно да не приспиваме целия процес, а само чакането за друг ресурс.