1.1 Оперативни системи

Wednesday, January 17, 2024 4:50 PM

Рачунарски систем састоји се од хардвера и софтвера који функционише по принципима Џон фон Нојмана.

Технички систем чине процесор, радна меморија, улазно-излазни уређаји, и остале тако да их све повезује магистрала.
Програмски систем чине апликативни и системски софтвер. Апликативни је највиши у тој хијерархији и служи за обављање специфичних задатака док је софтверски ту да нам омогући бољи интерфејс, односно представља повезаност хардвера и апликативног софтвера.

Оперативни систем је тај што треба да олакша коришћење и учини хардвер што употребљивијим и ефикаснијим. Задатак је такође и да заштити хардвер од корисника, односно његових програма.

Системски софтвер није само оперативни систем, него и алатке попут преводиоца, едитора, програма за сортирање и сличне (а нису део апликативног софтвера), без којих не би била иста удобност корисницима.

1.2 Основни концепти оперативних система

Wednesday, January 17, 2024 5:09 PM

Оно што је потребно да нам оперативни систем омогући уз што бољу ефикасност је:

- Управљање процесима
- Управљање меморијом
- Управљање улазно-излазним уређајима
- Управљање подацима
- Управљање мрежама

Управљање процесима односно програмима у извршавању, представља њихово креирање, извршавање, додељивање потребних ресурса, синхронизација, планирање редоследа извршавања.
Управљање меморијом ..., управљање излазно-улазним уређајима ..., и остало… --> страна 16

1.2.1 Функције језгра оперативног система

У језгру (кернелу) су смештене најважније функције које обезбеђују основне сервисе оперативност система. Управља на најнижем нивоу. Прво се учитава и последње искључује из система. Назива се и срцем система, у њему су дефинисана правила и дозволе за управљањем целокупног система.

1.2.2 Системски позиви

Услуге апликативним програмима оперативни систем пружа преко системских позива. Програм помоћу њих комуницира са језгром и врши осетљиве операције у систему. Системски позиви су скуп функција који представља интерфејс ка оперативном систему. Сви позиви су дефинисани тако да се зна како их користити и дефинисани су од потенцијалних штетних операција корисника по хардвер.

Постоје два режима рада, системски и кориснички (ограниченост). Неке фунцкије попут коришћење улазно-излазних уређаја, заштићених деловима меморије може само системски режим рада процесора приступити.

Апликативни програми се извршавају у корисничком режиму, док системски режим се користи за осетљиве операције. При потреби за осетљивим операцијама, систем се пребацује на системски режим и контролу даљу има оперативни систем. Кључан у таквим ситуацијама је језгро.

Сви програма, као и системски се раде на нивоу изнад језгра, али активности попут приступ хардверу, обављају се на нивоу језгра тојест у системском режиму.

Када апликативни програми желе да изврше системски позив, параметри системског позива се поставе на предвиђене локације у меморији, мења се режим рада и постају дозвољене све операција на ниову кернела. По завршетку, враћају се резултати а режим рада се враћа на кориснички.

1.2.3 Корисничко окружење

Подела им је на текстуална и графичка, односно боље речено линијска и екранска.

Линијска окружења су конзоле, терминали, командне линије, итд. које омогућавају да се управља оперативним системом помоћу куцањем текста - линијама.
Екранска окружења имају могућност да се управљају додатно помоћу миша, имају цео

екран на располагању а не само "линију".

1.2.4 Драјвери

Олакшавају комуникацију између улазно-излазним уређаја и система и њихово управљање њима. Дефинишу скуп дозвољених инстуркција. Прави се јединствен скуп дозвољених инструкција тако да се прави универзалан интерфејс за све уређаје исте класе.

1.3 Архитектуре оперативног система

Wednesday, January 17, 2024 5:38 PM

Типови језгара су:

- Монолитни системи
- Слојевити системи
- Системи засновани на микројезгру
- Хибридни системи
- Системи засновани на егзојезгру

1.3.1 Монолитни системи

Системи са монолитним језгром, односно да се у језгру налазе сви сервиси оперативног система заједно са драјверима интегрисани у један програм. Сви се покрећу у истом тренутку, у истом делу меморије. Приликом покретања оваквог система, језгро се у меморију учитава у целости као један извршни програм.
Функција може позвати једна другу без проблема.

Мана је лоша отпорност на грешке, баш јер се учитава целокупно као један програм. Ако једна ствар пукне, мора се поново покренути.

1.3.2 Слојевити системи

Овиме је оперативни систем изграђен у виду слојева, које се надограђују један на други. Слојеви се имплементирају тако да могу да користе искључиво функције првог слоја испод себе. Пројектовање слојева је компликовано баш због тих позивања функције нивоа испод.

Мана је неефикасност јер системски позив пролази кроз слојеве, где се при сваком пролазу прослеђују подаци, мењају параметри што га чини спорим. Не морају сви слојеви бити у језгру, него могу бити и у апликативном делу где нема пребацивања у системски режим.

1.3.3 Системи засновани на микројезгру

Заснива се на језгру са најосновнијим функцијама, зато је и микројезгро. Фукције које су до мало пре биле у језгру су сада у посебном нивоу меморије у којем свакако приступају у језгро са неког безбеднијег дела меморије. Тако се решавају проблеми током коришћења тих функција тако да се то неће одразити на функционисање језгра, баш јер је то у корисничком режиму рађено. Услуге са сличним задацима у оквиру оперативним система се називају серверски процеси, или само *сервери*. Тако се драјвери налазе на тим серверима а не на микројезгру. Сервер за управљање меморијом, улазно-излазним уређајима, итд...

Већи је степен сигурности у односу на монолитна језгра али су и спорија. Долази до кашњења јер се сервери и језгро не налазе у истом адресном простору и не извршавају у истом режиму. Прослеђивање порука између малој језгра и сервера чини кашњење.

1.3.4 Хибридни систем

Заснива се на хибриду монолитне и микројезгро арихитектуре. Неке веома битне и често коришћене функције се спуштају назад у језгро (за разлику од микројезгра) ради ефикасности и брзине, али се добар део задржава у корисничком моду изнад. Добро одабрати које спустити у језгро, а које држати изнад јер уколико буде варирало на једну страну више, биће ил монолитно ил микројезгро.

1.3.3 Системи засновани на егзојезгру

Постоје два облика, да програмери пишу програм и имају приступ хардверу или да то омогућава оперативни систем. Овакав егзојезгро системи су комбинација тога, идеја је да језгро обезбеди основне ресурсе и да апликацијама препусти рад са њима. Овиме програмери имају слободу у избору нивоа апстракције када је приступ хардвера у питању. Пошто је добар део функција сада измештен из језгра, знатно је мање него код осталих језгара.

1.4 Развој оперативних система и историјат

Wednesday, January 17, 2024 7:39 PM

Мултипрограмирање је концепт по коме у радну меморију треба сместити више програма (процеса) како би се искористила искоришћеност процесора (пошто до тада је искоришћеност била ниска). Меморија се дели у партиција у које би се учитавали програми. Програми раде и док је један запослен, остали раде улазно-излазне операције --> процесор је све време коришћен.

Дељење времена (time sharing) је концепт заснован на идеји дељењу рачунара на више корисника. Велики број корисника је захтевало рад на рачунару, па се десио проблем да физички процесор не можемо поделити, иако он може све то обрадити, тако да је смислено да се подели време за које корисник може слободно да користи и има процесор на располагању.

Мултитаскинг је само боље имплементовано мултипрограмирање. За мултитаскинг је јединица извршавања на процесору један таск који се не извршава нужно између његових улазно-изланих операција. Након извршавња процеса се ослобађа простор за следећи таск.

Мултипроцесирање или *мултипроцесорски системи* се односи на постојање више процесора у рачунарима који омогућавају рад више таскова.

2.1 Процеси

Wednesday, January 17, 2024 8:04 PM

Дефинише се као програм у извршавању. При покретању програма и учитавања у радну меморију, тај програм постаје процес. Процес је активан и оперативни систем је дужан да му обезбеди ресурсе за рад. Процес могу бити кориснички или системски, а њихово извршавање конкурентно или паралелно.

На системима са једним језгром се процеси извршавају конкурентно, оперативни систем чини рачунарсрки систем продуктивнијим где се ствара привид да се процеси извршавају у исто време, па се каже да се извршавају псеудопаралелно. Мултипрограмирање и остали концепти функционишу тако да смењују процесе које користе процесор тако да се сваки по мало извршава. Паралелизација на процесору са једним језгром је на слици на 56. страни где се извршавају 4 процеса на једном језгру и види се да не морају сви имати исто време коришћења процесора. Развијањем процесора се добило могућност да се више извршава у исто време.

Осим извршног кода и података, битне информације садрже и регистри а поготово бројач који зна докле је стигло са извршавањем процеса. Поред тога, информације о отвореним фајловима, дозволама и власнику процеса, итд.

Оно што је неопходно да оперативни систем обезбеди за ефикасан рад процеса су:

- Креирање и брисање процеса
- Управљање процесима
- Комуникација између процеса
- Синхронизација процеса

2.1.1 Процес у меморији

За разлику од програма који је само низ секвенци, пасиван је, процес је активан и извршава се по алгоритму имплементирано у програму. Као што је и речено, програмски бројач брине докле је стигло извршавањем, тојест која је следећа инструкција. Те инструкције се такође налазе у меморији са осталим подацима.

Програми када су учитани у радну меморију, тада већ процеси имају четири дела меморије:

- Стек сегмент
- Хип сегмент
- Сегмент података
- Код сегмент

Стек сегмент садржи локалне променљиве и параметре функција.
Хип сегмент садржи податке који су генерисани током извршавања програма, динамички алоцирани простор.
Сегмент података садржи глобалне променљиве.
Код сегмент садржи инструкције које процес треба да изврши.

2.2 Стање процеса

Wednesday, January 17, 2024 8:22 PM

Оперативни системи дефинишу стања у којима се процес може наћи у зависности од ситуације и сличног.
Нека од основних стања су:

- Нови --> процес је управо креиран, направљена је документација за њега и иде у спремно стање
- Спреман --> чека да му оперативни систем додели процесор
- Извршавање --> процес се извршава на процесору
- Чекање --> извршавао се али му је потребан ресурс да би се извршавао надаље, стога чека за тај ресурс
- Завршен --> процес је завршен и треба се избацити из система.

Прелазак процеса из једног стања у друго се дешава у следећим ситуацијама:

- Спреман --> Извршавање: када се процесор ослободи а оперативни систем одреди баш овај процес да буде први следећи за заузимање процесора.
- Извршавање --> Чекање: када процес током извршавања затражи ресурс који је тренутно заузет.
- Извршавање --> Спреман: када му истекне додељено време које је унапред одређено од стране оперативног система или ако оперативни систме донесе одлуку да неки други процес заузме процесор.
- Чекање --> Спреман: када процес дође до потребних ресурса, постаје спреман одакле поново стају у ред спремних и од оперативног система зависи када ће му дозволити да користи процесор опет.

Поред ових, постоји и *суспендовано* стање које могу бити на захтев корисника или самог система. Процеси у стањима спреман и чекање се могу суспендовати одакле они даље отпусте заузете ресурсе (не тражи ни оне који су му били потребни), често се пребаце на диск и чекају до престанка суспензије, а овако не заузимају радну меморију када се пребаце на диск.
До суспендованог спремног процеса долази уколико је превелики број процеса спремно тојест када је систем преоптерећен и тиме олакшава рад. Некад то систем уради како не би дошло до *заглављивања* система или уколико корисник зауставља програм због међурезултата.
До суспендовања процеса у стању чекања јесте због већ заузетих ресурса које има (на све то жели још) тако да се спречава заглављивање и убрзава рад система.
Међувремену из стања суспендован чека може доћи у суспендован спреман и када се створе услови пређе у само стање спремног процеса.

2.3 Контролни блок процеса

Tuesday, January 23, 2024 10:38 PM

Како би се лакше манипулисало процесима, сваки процес садржи ту своју динамичку структуру података са собом у којем се налазе потребне информације за идентификацију и управљање њима.

Садржај контролног блока гласи:

- *Идентификатор процеса* (*Process* ID - PID) --> јединствени број који сваки процес добија при креирању, сваки процес има свој засебан PID
- *Стање процеса* --> тренутно стање процеса
- *Програмски бројач* --> бројач који је помињан при извршавању програма, чува информацију о следећој инструкцији
- *Садржај регистра процесора* --> последње вредности које је имао у регистрима након губљења процесора
- *Приоритет процеса* --> однос важности тог процеса у односу на друге
- Адреса *меморије* где се налази процес --> показивач у меморији где се налазе сегменти процеса
- Адресе *заузетих ресурса* --> како би систем знао на којим локацијама да тражи потребне податке.

Поља контролног блока се деле на оне које се односе на процесе, меморију и фајлове.

Оперативни системи имају добру манипулацију тим контролним блоковима како би могли следеће:

- креирање контролног блока за нови процес
- уништањвање блока завршеног процеса
- мењање стања процеса
- мењање приоритета процеса
- избор процеса за извршавање

Појам *пребацивање контекста* значи када процес престане са радом, морају се замаптити његови параметри за рад, затим на његово место долази други процес који је добио процесор и његови параметри се учитавају.
Модул одговоран за ово се назива ***диспечер***. Задужен је за пуњење регистара процеса, пребацивање на кориснички режим рада и скок на позицију у програму да би се наставио.
страна 63 --> пример пребацивања контекста

2.4 Нити

Tuesday, January 23, 2024 10:53 PM

До сада је било да процес има своју једну линију извршавања али савремени оперативни системи подржавају нити. Поред ресурса и меморије које процес заузима и које нити деле, нити засебно могу да имају сосптевене ресурсе, свака поседује своје регистре, програмски бројач и стек а разликује их засебан id. Код сегмент, сегмент података и хип су заједнични за све нити једног процеса. Овиме се добија могућност *multithreading,* односно рад са више нити --> паралелно извршавање нити.

На почетку извршавања сваки процес добије меморијски простор и контролну нит која има задатак да иницијализује и креира остале нити, сама нит има карактеристике процеса као и то што деле неке ресурсе, називају се и лаким процесима.
страна 65. и 66. примери за нити, веб прегледач, текст едитор, како функционишу нити у томе, шта која ради, итд.

Предности нити су велике, штедња меморијског простора и времена су у питању. Нити деле меморију и неке ресурсе истог процеса чиме се знатно уштеди него независни процеси који би исто радили. Креирање је брже , као и пребацивање контекста јер се пребацују независности сваке нити а то су стек, регистри и програмски бројач.

У зависности да ли нитима управљамо са корисничког или системског нивоа, имају називе корисничке и нити језгра. Нити језгра имају право на приступ процесору и да се извршавају, стога постоји мапирање, а то су:

- више у једну
- једну у једну
- више у више

Више у једну значи да се више корисничких нити пресликава у једну нит језгра (слика на 68. страни).Тако је број нити језгра једнак и броју процеса у систему. Све нити се извршавају кроз једну нит, тако се нитима управља из корисничког режима без утицаја језгра. Која ће се извршавати се доноси у корисничком нивоу. И даље језгро ту нит језгра тог процеса гледа као и друге процесе. Постоје библиотеке које омогућавају манупилацију нитима у корисничком режиму.
Мана је ако се једна нит блокира, блокира се и низ језгра чиме су све нити блокиране, пошто само једна нит приступа језгру, пуцање једне нити се све блокира.

Једна у једну подразумева пресликавање сваке корисничке нити у једну нит језгра. Управљање нитима се скроз препушта језгру оперативнго система. Нити језгра се креирају спорије и има их мање, али језгро боље ради са нитима и има већи степен контроле.
Овиме се решава проблем претходног модела, пуцањем једне не блокира остале нити.
Мана оваквог модела је ограничавање броја нити језгра и самим тим исти број нити у корисничком режиму који је ограничен. Неки оперативни системи имају ограничен дозвољне број нити језгра због тежине прављења.

Више у више је хибрид, односно комбинација претходна два модела која у зависности од потребе процеса и могућности система мапира на нека два претходна начина. Број нити као и распоређивање корисничкин нити на нити језгра се одређује у корисничком нивоу, док остатак језгро решава. Број нити језгра које ће бити креиране зависи од конкретног процеса али и од броја процеса у систему.

2.5 Редови процеса

Tuesday, January 23, 2024 11:25 PM

Један од најважнијих задатака оперативних система јестеда обезбеде најефикасније извршавање процеса, да је искоришћеност процесора маскимална односно увек да обрађује неки процес али да се и они смењују најперфектније. Смењивање процеса током улазно-излазних операција, итд.

Неки троше више времена на процесору док неки са улазно-излазним операцијама. Процесор ће од свих процеса који треба да се изврше, изабрати оне који ће се одмах учитати у меморију и извршавати. Након почетне селекције, потребно је да се одреди редослед изабраних процеса, као и време за које могу да користе потребне ресурсе --> за то су нам битни *планери*.

Ред послова је почетни ред у којем се смештају покренути процеси. Садржи скуп свих процеса у систему.
Ред спремних послова садржи процесе који су изабрани по неком критеријуму од стране оперативног система. Процеси у овом рецу се налазе у стању ,,спреман'' и смештени су у главну меморију. Избор процеса за ред спремних зависи од потреба процеса, попут потребе за процесором или излазно-улазних операција, зато се прави добра селекција за што бољу ефикасност.
Оперативни систем води рачуна и о процесима који чекају на неке од уређаја, они су у стању ,,чекање''.
Редови су повезане листе контролних блокова процеса које нису увек организовани по FIFO (First in, First out).

2.6 Планери

Friday, January 26, 2024 9:33 PM

Постоје две врсте планера:

- Дугорочни
- Краткорочни

Уз њих се често имплементира и средњорочни планер.

Дугорочни планери - њихова фунцкија је да од скупа свих процеса који би требало да се изврше, а то је из реда послова, узме оне који ће да се активно укључе и почну са извршавањем. Од њих зависи највише јер морају да направе добар одабир за рад спремних процеса.
Краткорочни планери - од скупа процеса које је дугорочни саставио, одлучује који ће се извршавати и колико дуго ће добити процесор. Други назив му је и планер процесора.

Краткорочни планери се чешће позивају стога морају бити ефикасни при одабиру, док дугорочни због ређих позива није обавезно да буде што ефикаснији јер ипак од његове добре одлуке такође веома зависи ефикасност.

Због међусобног ометања и загављивања процеса, потребно је неке да суспендујемо како би се избавили из тога. Када се стекну услови, суспензија престаје и настављају са радом. Ова шема се назива *пребацивање (swapping).* За такве проблеме се зато уводи средњорочни планер који брине о суспендованим процесима.

2.7 Вишепроцесорски системи

Friday, January 26, 2024 9:42 PM

За разлику од размишљања на нивоу једног процеса (или чак коришћења нити које стварају илузију паралелности), увођењем више процесора олакшава рад и ефикасност.

Системи са нитима који раде на вишепроцесорским системима су најефикаснији али и најосетљивији, јер је захтевније да се искоришћености употреби до максимума. За почетак синхронизација нити истог процеса које се могу поделити на више процесора, што је наравно битно ако је редослед тих нити неопходан да буде како треба.

Овде код вишепроцесорских система постоје следећи начини распоређивања процеса:

- симетрично распоређивање
- асиметрично распоређивање

Симетрично распоређивање подразумева да су процесори равноправни и да се процеси на њих распоређују на било који од алгоритма за распоређивање. Код овог приступа сви процесори деле меморију, деле приступ за улазно-излазне уређаје и оперативни систем их третира поптуно равноправно.

Асиметрично распоређивање функционише тако што неки процесори могу бити задужени за одређене функције, постоји један главни којем се поверава извршавање кода језгра оперативног система и улазно-излазне операције. Остали процесори извршавају корисничке процесе додељене од главног процесора. Испада да овакав начин није најефикаснији, тако да је симетрично распоређивање испало боље.

При распоређивању процеса, могуће је спремити један заједнички ред за све процесоре или појединачне редове за процесоре.
Приступ заједничким редом доводи до равномерног оптерећења процесора где чим процесор постане слободан, добиће нови процес да ради, где може доћи до тога да два процесора узму исти процес. Затим при повећању броја процеса, долази до стварања *уског грла.*
Код другог приступа постоји могућности да неки процесор има у свом реду пар процеса док је неки процесор незапослен, што се решава увођењем *балансирањем оптерећења (load balancing).* То је поступак да се што равномерније расподеле процеси процесорима, при чему ту постоји два начина:

- *преношење миграција (push migration)*
- *преузимање миграције (pull migration)*

Преношење миграције је случај где ће се од оптерећеног процесора, пренети процеси на друге мање запослене или безпослене процесоре. Сложеност овога није занемарљива јер је потребни следеће: алгоритми за одређивање оптерећености свих процесора, проналажење најмање заузетог (оптерећеног), избог процеса за пребацивање, итд. То се може десити ако оптерећеност пређе неку границу или у одређеним временским интервалима.

Преузимање миграција подразумева да се из редова заузетијих процесора пребаце процеси у мање заузет процесор. Потребни алгоритми би били за проналажење најслободнијег и најзаузетијег процесора, за одабир процеса за пребацивање --> такви процеси би били они са високим приоритетом и они коју се кратко извршавали.

Код балансирања оптерећења долази до тога да се процес не изврши у потпуности на једном процесору чиме долази до веће сложености извршавања. Тада се меморија једног процесора празни и та иста мора да се пребаци у други процесор. Због овога се уводи да се процесу додели

афинитет ка одређеном процесору (*processor affinity).* Он се углавном развија према процесору на коме је и почело првобитно извршавање.
Афинитет према процесору може бити *слаб* (*soft*) или јак (*hard*).
Слаб афинитет не обећава да ће процес бити извршаван само на једном процесору, ако је потребно, промениће се на други процесор и развити афинитет према том новом.
Јак афинитет подразумева да оперативни систем допушта да се процесу прецизно одреди подскуп процесора кна којима може да се извршава. Многи системи подржавају оба афинитета.

Афинитети и балансирање су у супротности тако да при пројектовању система се треба одлучити за један принцип.

2.8 Распоређивање процеса

Friday, January 26, 2024 10:10 PM

Од њега доста зависи ефикасност ситема, због те брзе одлуке о промени стања процеса који се налазе у меморији.

Код једнопроцесорских система је то један ред где нема много размишљања али када се на то укључе нити и вишепроцесорски системи, постаје другачије и захтевнија је боља одлучност, наравно и брза.

Следећи алгоритми су описани за једнопроцесорске системе:

- *искоришћеност процесора* --> искоришћеност у процентима у одређеном временском периоду
- *пропусна моћ* --> број процеса у јединици времена
- *време обраде* --> време од покретања процеса до краја извршавања (рачунајући сва стања)
- *време чекања* --> време које процес проведе чекајући
- *време одзива* --> време од пријављивања процеса до првог произведеног излаза програма.
- *количник дужине времена одбраде и трајања извршавања процеса* --> добар параметар за описивање ситуације у систему

Под оптимизацијом процеса, решења за то треба да имају:

- максимала искоришћености процесора
- максимална пропусна моћ
- максимално време обраде
- минимално време чекања
- минимално време одзива
- Конвергенција количника дужине времена обраде и времена извршавања процеса ка 1

2.9 Алгоритми планирања

Friday, January 26, 2024 10:46 PM

Алгоритми за планирање при којима има и прекидање процеса, односно пребацивање контекста које се дешава на основу одлуке планера, односно система.

2.9.1 FCFS алгоритам

First Come, First Server - у зависности од реда како дођу процеси, по том реду се и извршавају, без прекидања јер би тако прекршо на то што се заснива. Концепт је FIFO (First In, First Out). Није ефикасан јер се ред чекања процеса може повећати због једног који може бити јако дуго на процесору, наравно без прекидања као што смо рекли.

ТАБЕЛЕ НА 84. СТРАНИ

2.9.2 SPF алгоритам

Shortest process First - фаворизују се процеси чији је следећи захтев за процесором (процесорским временом) мањи. Одређује се време потребно за извршавање следећег посла процеса, а на процесор долази процес коме је тај посао са најмањим временом. У случају два процеса са истим временом, FCFS одлучује између та два.

Може бити имплементиран са или без прекидања процеса на процесору. Када је дозвољено пребацивање и када дође процес са мањим захтевом, тада долази до пребацивање контекста и нови процес добија процесор за извршавање. ТАБЕЛЕ НА 85. и 86. СТРАНИ

Оно што се добија прекидањем јесте да ће средње време чекања бити мање, што јесте ,,позитивна'' ствар, али наравно мане су следеће:
Одређивање дужине следећег захтева са процесором односно процена дужине трајања тог следеће активности процеса на процесору (до улазно-излазне операције или краја процеса). На то утиче дугорочни планер који задаје временску границу потребну да се изврши посао. Ефикасност система зависи од реалног предвиђања времена да се посао изврши, на шта ће они који буду одређени да су краћи - буду фаворизовани. Краткорочни тек не може одредити то. За предвиђање дужине следеће активности се користи *експоненцијална средина* претходних активности и претходних процеса потребног времена. У преводу, следећа активност ће бити слична дужини као прошлим активностима истог процеса. Дужина трајања следеће активности се може изразити:

$$T_{n+1} = \alpha T_n + (1 - \alpha) t_n$$

-t_n је време које је било потребно да се n-та активност процеса изврши на процесору.
-T_n је очекивано време које је било за одређивање n-те активности процеса на процесору.
Ако је α ближе јединици, онда се у обзир више узима претходна процеса, а у супротном ће утицати стварна дужина последње активности.

2.9.3 Алгоритам са приоритетом

Алгоритам са приоритетом подразумева да се сваком процесу додели његов приоритет који се заснива у односу на нешто, примера ради FCFS има приоритет како је ко дошао од процеса, SPF има приоритет процес који има најмање очекивано време да изврши следећу активност.

Приоритети се дефинишу интерно или екстерно. Интерно дефинисани узимају неку мерљиву јединицу за одређивање, на то утиче временске границе, број отворених фајлова, меморијски захтеви, просечно трајање активности на процесору, просечно трајање улазно-

излазних операција и количник последње две наведене вредности.
У преводу, процеси који користе улазно-излазне операцијаме би требало да имају приоритет јер им је процесор потребан за јако кратко време, јер често и чекају блокирани на неки уређај. У супротном би дуго чекали на процесор а улазно-излазни уређаји би чекали и били незапослени.
Ескетрни приоритети су спољашњи, то је нпр. важност пројекта, временски рокови, важност клијента.

Највећа мана овог алгоритма је ,,изгладњивање'' процеса. Наиме, ако процес чека свој ред али упорно долазе процеси са већим приоритетом. Ово се решава тиме да се процесу повећа приоритет како дуже проводи време на чекању за процесор.

Постоје верзије са и без прекидања, што значи да ако дође процес са већим приоритетом, текући се прекида и нови долази на процесор и извршава се.

2.9.4 Кружни алгоритам

Кружни алгоритам (Round Robin) је опслуживању рачунара, слично нешто као што је *дељење времена (Time-sharing)* у питању. Сваки процес добије процесор на неки квантум времена. Ако му квантум прође, прекида се, ако се свакако изврши, долази у оба случаја следећи процес из реда спремних процеса. Прекинути се ставља на крај ако се није до краја извршио. На n процеса, за квантум q, сваки процес добија 1/n по q у једном кругу.
За превелики квантум би било исто што и FCFS, за премали квантум се назива *дељење процесора* што би било да се сваки од n процеса у реду има свој процесор n пута спорији него у систему.

Због пребацивања контекста након сваког квантума, квантум мора бити знатно дужи од тога да би чак имало смисла. Имати дужи има смисла али не претерати јер онда добијамо FCFS.

2.9.5 Редови у више нивоа

Редови у више нивоа је идеја да се добијени ред спремних процеса подели у више редова са различитим приоритетима. Процес се припаја реду на основу неког критеријума, а унутар мањег реда такође има неки свој алгоритам. Процес добија процесор ако су редови вишег приоритета празни.

2.9.6 Редови у више нивоа са повратном везом

Редови у више нивоа са повратном везом је модификација претходног приступа. Нека од разлика је допуштање да се процес креће кроз различите редове, тиме има већи/мањи приотитет. Идеја је да се током извршавања издвоје процеси са различитим трајањем активности на процесору, тако да се процеси који троше превише времена сврстају у редове са нижим приоритетом. Овиме се процеси са улазно-излазним операција се сврстају у редове вишег приоритета. Сваки процес у вишем реди искоришћава квантум времена q и након што га искористи, ако се није завршио, губи приоритет односно премешта се и нижи ниво.

2.10 Управљање процесима у системима за рад у реалном времену

Friday, January 26, 2024 11:41 PM

Овакви систем се пројектују за одређену примену, када постоје временска ограничења за извршавање унапред дефинисаних послова.
Могу се поделити на *чврсте (hard) и лабави (soft).*

Чврсти системи за рад у реалном времену су такви да један процес у целокупном систему има фиксни крајњи рок за ивршавање који се мора испоштовати. Ако дође до прекорачења, систем вероватно отказује и решења су бескорисна. Неки од система су: уређаји у авионима, одбрамбени системи, авио саобраћај, уређаји у возилима, електрана.
Лабави системи јесте да је рок пожељан, али није од пресудног значаја за функционисање система. Процесима се одреде рокови и прекорачење утиче на ефикасност али не и на функционисање - прекорачење се толерише у извесним границама. Ови систему су у банкарству, уређаји за резервисање авио карата, аудио-видео уређаја, итд.

Процеси могу бити периодични и неприонични (асинхрони).
Периодични процеси карактерише понављање после тачно одређеног временског периода.
Непериодични процеси обично имају рок када да почну или да заврше са извршавањем, али могу и почети са извршавањем између та два рока.

2.10.1 Распоређивање процеса у системима за рад у реалном времену

3. Конкурентност и синхронизација процеса

Saturday, January 27, 2024 12:27 AM

Проблеми који настају приликом промена процеса или нити, приступању заједничкој меморији или слично, долази до потенцијално нежељених резултата, наравно циљ је да се то спречи и омогући на ефикасан начин.

Ситуација у којој од редоследа извршавања корака зависи резултат, назива се *трке за ресурсе*. Први пример јесте увећавање заједничке променљиве, нпр. за 1, та операција није атомична што значи дасе извршава из три инструкције уместо једне. Пример у књизи на 106. страници.

3.1 Критична секција

Saturday, January 27, 2024 12:48 AM

Део програма који чине заједнички подаци. Треба обезбедити да у једном тренутку само један процес може да јој приступи.
Прва идеја је да се онемогући прекидање процеса који приступа критичној секцији. Систем без прекидања је редак а и губи се идеја мултипрограмирања без тога.
Оно што би било решење за критичну секцију треба да задовољава следеће услове:

- *Узајамна искључивост* --> два процеса не могу бити у исто време у критичној секцији
- *Услов прогреса* --> процес који није у критичној секцији и не жели да уђе, не смета осталима да уђу у њу.
- *Услов коначног чекања* --> не сме се догодити да неки процес у недоглед чека да приступи критичној секцији.

3.2 Решења заснована на активном чекању

Saturday, January 27, 2024 12:49 AM

Чекање у петљи док се не створе услови за приступ критичној секцији.
Читати из књиге, нема шта специјално издвајати.
Проблеми се дешавају ако дође до прекида процеса, а у том тренутку смо поставили неку вредност и можда се угаси, а баш тада други процес чека у петљи, итд.

3.2.1 Стриктна алтернација

Заснива се на коришћењу променљиве na_redu којом се одређује који од два процеса има предност.
СТРАНА 113.
Од три услова које треба да буду испуњени, овај испуњава узајамну искључивост. Ако један процес има предност али не жели да уђе у критичну секцију, блокира други процес тиме и онемогућава га --> услов коначног чекања, не зна се ни кад ће ни да ли ће.
Мана је што је намењен за два процеса, али је погодан за наизменично приступање дељеним ресурсима.

3.2.2 Декеров алгоритам

За решење су потребне три променљиве:

- zeli_proces_1 --> њом први процес најављује да жели да уђе у критичну секцију
- zeli_proces_2 --> њом други процес најављује да жели да уђе у критичну секцију
- na_redu --> води рачуна о томе који процес је на реду да уђе у критичну секцију

Странаица у књизи 114.
Овим алгоритмом се искључује да приступе критичној секцији у исто време, не може доћи до заглављивања. Може ако један процес прође два пута чиме се нарушава принцип праведности.
Овај алгоритам је намењен искључиво када два процеса конкуришу за исте податке.

3.2.3 Питерсонов алгоритам

Основна идеја је да када процес жели да уђе у критичну секцију, он ,,џентлменски'' уступа предност другом процесу. А та предност важи уколико други жели да уђе у критичну секцију, у супротном први процес не чека на другог.

Књига, стрна 116.

3.2.4 Лампортов (пекарски) алгоритам

Уопштење Питерсоновог алгоритма за n процеса представља *Лампортов алгоритам.*
Као услуживање у пекари, добијање бројева тако да они са мањим су услужени пре.
Критична секција је и добијање бројева ако два процеса желе ући, онда ће онај с мањим индексом бити први услужен.

Уводе се следеће ознаке:

- (A, B) < (C, D) ако A < C или A = C и B < D.
- $\max(A_1, A_2, \dots, A_n) = K$, при чему је K такво да је $K >= A_i$, i = 1, 2, ..., n

Као глобалне променљиве се уводи целобројни низ (broj[n]) и низ логичких променљивих (uzima[n]) и оба низа иницијализовати на нулу.

Овај алгоритам задовољава сва три услова, ради за произвољан број процеса и ради на системима са више процесора.

3.2.5 Хардверска решења

Идеја је да се направе машинске инструкције којима се штити критична секција, да се извршавају атомично. Користе се следеће три инструкције:

- TAS (Test And Set)
- FAA (Fetch and Add)
- SWAP (замена)

TAS користи две променљиве, A = TAS(B) и ради тако што A добије вредност B а B добије вредност 1.
FAA исто користи две променљиве, FAA(A,B) функционише тако што A добије вредност B, а B добије вредност A + B.
SWAP такође са две променљиве, SWAP(A,B) и ретултат је атомична замена вредности A и B.

Обично постоји једна променљива којом се штити улаз у критичну секцију док остали процеси активно чекају да се улаз ослободи.
АЛГОРИТАМ на страници 119.
Мана је што је засновано на хардверској подршци, односно да процесор има уграђену ту инструкцију. Може бити корисно на прозивољан број процес осим тога што улазак зависи од среће ткд може теоријски доћи до изгладњивања процеса.
Ту се може обезбедити гаранција да ће коначно дочекати улазак у критичну секцију, модификацијом алгоритма. Декларише се глобални низ *ceka[n]*, иницијализовати све на нулу и променљива *zauzeto* која такође се поставља на нулу.
Алгоритам је на страни 120. Мало је спорији због додатне провере али је засигурно да ће сваки процес дочекати свој ред.

Заштита се може обезбедити коришћењем SWAP атомичне операције.

3.3 Решење за заштиту критичне секције без активног чекања

Saturday, January 27, 2024 3:20 PM

Ова решења се састоје без активног чекања, заправо ефикаснији начини да се уђе у критичну секцију, без беспотребних циклуса за улазак. Најпознатији модели су:

- семафори
- критични региони
- монитори

3.3.1 Семафори

Осмислио Едсгер Дијкстра, исти онај из ДС2 алгоритам.
***Семафор* апстрактни тип података**, структура која блокира процес на неко време док не стекне услове да буде пропуштен. Поред стандардних операција прављења и ослобађања, постоје и P и V операције, односно *wait* и *signal.* Конкретно P и V су Proobeer (покушај, пробај) и Verhoog (увећај, инкрементирај).

Операција P(S) смањује вредност S за 1 и ако је ненегативна операција (ако је био 1), тај процес се пропушта, у супротном се процес блокира и ставља у ред за чекање.
Операција V(S) повећава вредност S за 1 и вредност S није позитивна, значи да постоје блокирани процеси у реду за чекање тако да након последњег процеса долази на вредност 1 --> позитиван броја, значи нема процеса у реду.

ДИЈАГРАМИ

Ови семафори су *општи* или *бројачки* јер дозвољавају негативне вредности, поред њих имају и *бинарни* који имају само вредност 0 и 1.

СТРУКТУРА --> ВРЕДНОСТ & ЛИСТА БЛОКИРАНИ ПРОЦЕСА

Процеси пре уласка тестирају семафор и ако је 1, пролазе, у случају 0 значи да је неко већ у том критичној секцији. По завршетку критичне секције, процес извршава операцији V(S) тако да први следећи процес из реда или који желе ући, може приступити, ако не, S има вредност 1.

Пошто су семафори структуре, оне такође садрже и показивач на листу која чува податке о процесима који су блокирани на том семафору.

СВЕ ИМПЛЕМЕНТАЦИЈЕ СУ У КЊИЗИ

Може се имплементирати која ће се функција пре десити, стави се семафор на 0, функција1 се уради и дигне на 1, да би тек код функција2 пробала P(S) и да започне тек функцију након што је пуштено.

Имплементација кода са једним бројачким и два бинарна семафора:

3.3.2 Употреба семафора

Решавање следећих проблема:

- Проблем произвођача и потрошача
- Проблем филозова који вечерају

- Проблем читача и писача

3.3.2.1 Проблем произвођача и потрошача
3.3.2.2 Проблем филозова који вечерају
3.3.2.3 Проблем читача и писача

3.3.3 Критични региони

Бринч Хансен и Тони Хор уводе критичне регионе. *Критични региони* представљају имплементацију критичке секције на вишем програмском језику.
Треба дефинисати променљиву коју деле више процеса и ознаком *регион* обезбедити ексклузивни приступ тој променљивој у оквиру низа наредби које се налазе у том региону.

у књизи 139. страница

Оно што критички регион гарантује, је то да извршавање наредби у том региону, било који други процес не може да приступи уколико су придрођени тој истој дељеној променљивој. Њој само могу приступати искључиво у оквиру неког од региона где је заштићена.

Овим начином се решавају проблеми критичне секције али је синхронизација мало тежа. Напреднији концепти се заснивају на такозване условне регионе, које се активирају само у случају да је услов тачан, итд.
Ако је услов испуњен, процес има право да приступи региону (ако већ неки други не приступа) и у њему има приступ променљивој *v*. Ако услов није испуњен, улазак у регион се одлаже док услов не буде задовољен и док у *v* не буде ниједан други процес.

Додатне модификације за синхронизацију које допуштају да се привремено прекине рад у критичнј секцији и провери нови услов (коришћењем наредбе *AWAIT*). Што би значило да има додатну проверу услова.

Пример 141. страница
Када процес уђе у критични регион, извршава се првих *k* наредби које могу бити и празне. Затим се израчунава вредност *uslov,* па ако је тачна, онда се извршавају к+1 и наредне наредбе. У случају а је *uslov* нетачан, одлаже се наставак, процес напушта критични регион, одлаже извршавање наредби док услов не буде тачан и при томе да ниједан други процес није тренутно у свом истом критичном региону.

3.3.4 Монитори

Монитори су највиши ниво када је у питању заштита критичке секције без активног чекања. Представљају конструкције програмских језика, слично као класе у ООП. Главна одлика монитора је да се у оквиру њега у једном тренутку може бити активан само један процес, чиме се обезбеђује заштита.

За синхронизацију се користе
специјалне условне (*condition*) променљиве.
Над оваквим променљивима се користе операције *wait* и *signal* чија је синтакса: x.wait(), x.signal. Процеси се тиме могу блокирати или одблокирати сигналом из другог процеса. Овиме је решено и приступи критичној секцији и синхронизацији.

Ако се шаље сигнал а ниједан процес не чека на сигнал, он ће се само изгубити и тјт. Са једним сигналом се тачно један процес пробуђује.

Функције у монитору може позванти један процес, док је он у монитору (и користи те функције), ниједан други не може позвати те функције.

Монитори су виши ниво, за разлику од семафора који су на нивоу оперативног система. Предност монитора је у сигналима *wait* и *signal*.

Монитори се могу реализовати уз помоћу семафора. Потребан је *mutex* за ексклузиван приступ монитору*,* семафор *procesi* који се користи за синхронизацију процеса који су у монитору и променљива *brojac_procesa* која чува информацију о броју таквих процеса.

страна 145.

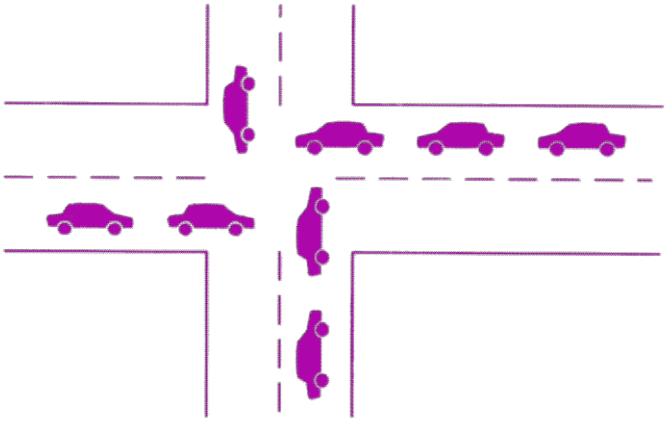
3.3.4.1 Решење проблеам филозова који вечерају уз помоћ монитора

а.

4. Заглављивање

Monday, November 20, 2023 12:15 PM

Два или више међусобно зависних процеса који чекају на ресурсе које никад неће добити, назива се заглављивање (deadlock). Ово заглављивање је трајно, чиме се сматра да је и систем цео заглављен.



Дат је пример са раскрсницом аутомобила који поштују правило десне стране, на овакав начин долази до заглављивања. Такође пример са 5 филозофа - уколико би свако узео по један штапић (5 филозофа - 5 штапића), дошло би до заглављивања. Ниједан од њих неће спустити штапић, а у исто време сваки од њих чека да неко пусти штапић.

Појмови који ће бити коришћени, као и тема поглавља су: ***живо блокирање*** (livelock) и ***изгладњивање*** (starvation).

Ситуације где процеси нису блокирани, али не напредују у извршавању се називају ***живо блокирање.*** Пример из живота је када покушавамо да се мимоиђемо са неким, па када се померамо у исту страну лево-десно, не стојимо у месту али не напредујемо. Пример са раскрсницом би билода са свих страна дођу у исто време по 1 аутомобил, када се заглаве, одлуче отићу уназад и поново напред да би прошли - пошто раде у исто време, поново се заглављују.

SLIKA 4.2

Изгладњивање је ситуација у којој процеси напредују у систему али неки процеси доста дуго не напредују. Такве ситуације су да нису трајне као живо блокирање и заглављивање, овде заправо ће доћи тренутак да се ти процеси наставе, али то је произвољне дужине.

У системима који подржавају мултипрограмирање може лако доћи до заглављивања. Ако процес дође у стање *блокиран* чекајући на ресурсе које никад неће добити, сматра се да је процес заглављен. Пошто се паралелно све дешава, процеси траже ресурсе и онда међусобно узму ресурсе тако да нико није у стању да искористи своје и преда другима, заглаве се међусобно. Ресурси могу бити хардеверски попут диска, меморије, процесора, скенера, итд. и софтверски попут семафора, фајлова, слогова у базама података, итд.
У обе групе постоје они који када се користе, не могу се прекидати док се не искористе до краја, док не заврше своју радњу. За процес је важно да добије неопходну количину времена и меморије за извршавање.

До заглављивања долази са ресурсима који када при започињању коришћења морају бити

искоршћени, нпр. критична секција --> процес који јој приступа не сме да буде избачен док не заврши са њом. Затим штампач као најлогичнији ресурс који не сме бити прекидан.

Хофманови услови:

Да би дошло до заглављивања, потребни су да се испуне истовремено следећа четири услова:

- *Узајамно искључивање* --> тачно један процес може користити ресурс, остали чекају на ослобађање
- *Чекање и држање* --> процес држи ресурс али не ради јер тражи неки ресурс додатан да би радио, који нису слободни
- *Немогућност прекидања* --> оперативни систем не може да одузме ресурс процесу којем је већ дао, да би доделио другом, него сам процес који их користи мора да их отпусти.
- *Кружно чекање* --> међусобно траже ресурсе једни од других и нико тиме не напредује

Пошто треба да се испуне сви наведени услови, неће доћи ако се барем један уклони.
Постоје тако четири групе за мере које се примењују:

- ***Спречавање*** --> системске мере које се предузимају на почетку рада како би се искључиле могућности за било какво заглављивање
- ***Избегавање*** --> динамичке мере које се предузимају како би се систем водио кроз стања која обезбеђују да не дође до заглављивања. За разлику од првог, ово је либерална мера.
- ***Детекција и опорављање*** --> дозвољава појављивање, детектује га и третман тих пронађених заглављених процеса и покушава да спасе део резултата при томе.
- ***Непредузимање било чега*** --> ово се користи код система где до заглављивања ретко долази или када није важан посао. Ако дође, поново се покреће процес или систем и то је то.

4.1 Мере за спречавање заглављивања

Sunday, January 28, 2024 3:06 PM

Мере за спречавање заглављивања су статичке, системске мере које нису алгоритамски захтевне тако да у старту имају јасно дефинисана правила за извршавање процеса, чиме се елиминише бар један од услова од 4 која постоје.

4.1.1 Превенција чекања и држања

Главно решење је да процес не може да држи ресурсе и да у исто време тражи нове ресурсе. Постоје два приступа са којима се решава ово.

Први приступ ради тако да процес се извршава тек након што сакупи све ресурсе. Лоша страна је да док не почне са извршавањем, он их држи за себе и онемогућава остале да их користе док их он још не користи.
На пример, ако процес треба да прочита фајл са флеша, сними на диск и одштампа, то је три ресурса. Зашто би он држао штампач док он чита са флеша и снима на диск.
Често се ни не зна који ће сви ресурси бити потребни пре коришћења тако да отежава употребу овог алгоритма.

Други приступ подразумева да процес тражи ресурсе када му затребају, али да пре него да добије, првобитно врати све ресурсе које је до тад користио и тек онда оперативни систем одлучује да ли ће му их доделити или неком другом процесу. У примеру за флеш, диск и штампач би штампач држао тек кад треба да штампа, а то када би га добио, зависи од одлуке система.

Овим мерама се спречава у потпуности, али искоришћеност је мања и може чак доћи до *изгладњивања* јер систем не даје неком процесу ресурсе и никако се не може завршити.

4.1.2 Елиминисање немогућности прекидања

Идеја се заснива на томе да када се потребе процеса за ресурсима не могу ипсунити, отпуштају сви ресурси које он држи. Када процес држи неке ресурсе а захтева неке које не може одмах добити, он све оптушта и чека док не добије их све назад заједно са оним које је тражио и тако наставио са радом.
Захтевнији процеси ће подуже чекати да би се завршили.

4.1.3 Превенција кружног чекања

Да не дође до кружног чекања, обично се користе решења која подразумевају означавање, тојест енумерацију типова (класа) ресурса.
Идеја је да се сваком ресурсу долели један број, тако да се процесу који држи ресурсе додељују само ресурси који имају већи додељени број од бројева његових ресурса.
Може тражити ресурс нижег броја тек након што отпусти све ресурсе.

СЛИКА 4.6 на страници 162.

Потешкоћа је како енумерисати ресурсе, са којим бројевима, као и поновна енурмација сваки пут када се појави нови ресурс.

Овиме се сигурно онемогућава да дође до кружног чекања.

4.2 Мере избегавања

Tuesday, November 28, 2023 12:39 PM

За разлику од мере спречавања, које од старта елиминишу неке случајеве заглављивања, ово су динамичке мере које проверавају стање у систему у одређеним моментима, алгоритми су већи али ефикаснији јер се ресурси боље користе.

4.2.1 Банкаров алгоритам

Алгоритам је 1965. године предложио Edsger W. Dijkstra. Назив је добио по истом принципу давању зајмова банкама. Банка има ограничене ресурсе, а корисници знају колико је укупно ресурса потребно да би завршили свој пројекат. Банка ће некима изаћи у сусрет (кроз један или више кредита) како би корисник завршио пројекат. Корисник који је добио онолико колико му треба и завршио пројекат, враћа новац назад на располагању банци како би она даље давати осталим корисницима да би и они завршили своје пројекте. Банка мора добро проценити коме дати кредите, јер уколико дође да нико не може завршити свој пројекат, а банка је остала без новца, корисници неће моћи да заврше своје пројекте.
У складу с овим, пре почетка ивршавања сваког процеса се зна колики је максимални потребан број ресурса, так да ће се динамички испитује покушај доделе како би се обезбедило да никада не дође до кружног чекања.
Променљиве су број расположивих, додељених ресурса и маскимални захтеви процеса.
Стање система је безбедно уколико систем може да у одређеном редоследу додељује ресурсе процесима тако да ће у сваком неки завршити са радом и ослободити ресурсе за остале процесе, док се сви не изврше.
Редослед по ком се врши подела ресурса процесима се назива *безбедна секвенца.*

ДЕФИНИЦИЈА - Секвенда процеса је безбедна за текуће стање акко: $P_1,P_2,P_3,\ldots,P_n$ је секвенца таква да потребе i-тог процеса могу задовољити слободни ресурси и ресурси претходних процеса P_j ($j < i$).
Ако потребни ресурси нису тренутно доступни, чека се да се P_j процеси изврше, чиме се ослобађају ресурси.

ДЕФИНИЦИЈА - Систем је у безбедном стању акко постоји безбедна секвенца.
Ако стање није безбедно, то значи да потенцијално систем може дати ресурсе процесу који се неће завршити, али може и да не дође до заглављивања.

пример 4.1
Нека систем има 10 истанци неког ресурса:

Процес	Максималне потребе	Тренутно стање
P_1	7	3
P_2	5	2
P_3	9	2

Секвенца која је безбедна је $< P_2, P_1, P_3 >$, може се приметити да систем може пружити 3 ресурса другом процесу, чиме му се враћа 5, тако на даље може први процес завршити своје...

Идеја је да се ресурси додељују процесима тако да се систем спроводи кроз низ безбедних стања како би се избегла могућност заглављивања. Потребно је да свима буде угођено (али и да остане безбедно).

По захтеву процеса за добијање ресурса врши се провера да ли ће тај зајам касније вратити и бити на располагању другим процесима, такође да ли ће зајам утицати на завршавање других процеса.

Алгоритам 4.1 Провера стања

```
//иницијализација
radni raspoloživo;
gotov[i] = 0, za i = 0, 1, ..., n -1;
sledeći proces_u_sekvenci:
//Provera da li postoji proces koji može biti zadovoljen raspoloživim
 //resursima
IF (ne postoji i takvo da je gotov[i] == 0 i potrebno[i] < radni)
        idi na kraj;
ENDIF
//Pronalazi se proces koji može biti zadovoljen raspoloživim resursima
 Odaberi prvo i takvo da je gotov[i] == 0 i potrebno[i] < radni;
//Radnom skupu se dodaju resursi procesa koji ima mogućnost da se izvrši radni
radni + dodeljeno[i];
gotov[i] = 1;
idi na sledeći proces_u_sekvenci;
kraj:
//Ako su svi procesi imali mogućnost sa se izvrše - stanje je bezbedno
IF (gotov[i]== 1 za svako i)
        bezbedno stanje = true;
ELSE
        bezbedno stanje = false;
ENDIF
```

У књизи се врте примери и псеудо кодови, сви се могу разумети помоћу текстова горе.

4.3 Детекција заглављивања

Tuesday, November 28, 2023 3:48 PM

У оваквом концепту се не гарантује да неће доћи до заглављивања него да ако дође, шта се примењује да се решимо. Откривање може бити активно и пасивно. Активан приступ користи алгоритам откривања проверавајући да ли је дошло до заглављивања, учесталост му је одређена фреквенцијом заглављивања.

Стање система се може приказати графичким путем, где су круговима приказани процеси а правоугаоницима ресурси (више инстанци ресурса су кругови унутар праоугаоника). Стрелица од ресурса ка процесу значи да је тај ресурс коришћен од тог процеса, а обратно је да процес тражи тај ресурс.

Идеја ове детекције јесте да се уочи циклус у графу. Граф који треба посматрати је такав да се уклоне ресурси и прикаже заправо који процес чека други процес (користи ресурс који другоме треба). Стога, ако постоји циклус, значи да постоји заглављивање. Сложеност *је O(n*), где *је n* број грана графа.
Алгоритам се заснива на тражењу процеса који могу бити задовољени са расположивим ресурсима. Тако да рад овог алгоритма би изгледао тако да како процеси заврше своје радње, избацују се са графа. Ако се не избаце сви процеси до краја, дошло је до заглављивања.

4.3.1 Опоравак заглављивања

Након утврђивања да је дошло до заглављивања, потребно је да систем одреагује и уради нешто поводом тога. Најједноставније је да систем блокира процесе, одузме им ресурсе и кориснику понуди да одлучи шта ће са њима (у пракси најчешће се користе аутоматизоване операције). Користе се два приступа: прекидање једног или више заглављених процеса, други је одузимање ресурса заглављеним процесима.

Први алгоритам подразумева да се уклоне сви блокирани процеси чиме се губи дотадашњи резултати тих процеса. Мало бољи алгоритам функционише тако што уклања један по један процес да се којим случајем систем "опорави". У таквом случају треба одабрати који процес да се уклони. Узимају се следећи критеријуми:

1. Приоритет процеса - једноставно прекинути процес најнижег приоритета
2. Колико дуго се процес извршава и колико му је остало до краја - прекидати процесе који су тек почели са извршавањем, а оне што су при крају да се одлаже до последњег тренутка.
3. Које ресурсе процес користи и који су му потребни да заврши са радом - процеси који задржавају много ресурса ће бити уклоњени или они који користе ресурсе који су такође потребни и другим процесима.

Идеја одузимања ресурса је таква да ослобађање ресурса од неких процеса може отклонити заглављивање (оставити процес у стању да је касније могуће наставити са њиховим радом). Најгоре је да процес мора поново од почетка да се извршава, док је боља опција да се нађе тачка од које се може наставити када се стекну услови, уз то да се сачувају резултати пре одузимање ресурса.

5. Управљање меморијом

Wednesday, November 29, 2023 3:19 PM

Основни ресурс рачунарског система. Свака процесорска инструкција се мора наћи у меморији, а за извршавање већине је потребни су подаци из меморије који се доносе у процесор или из процесора пребацити у меморију. Проблемом меморије се баци Менаџер меморије.

Меморија је било који уређај који трајно ил привремено чува податке. На основу чувања, имају оне које се чувају док је укључен рачунар и оне које чуваку када је искључен (трајно чување). Подела се дели на основу брзине пруступа тој меморији, а то су следеће четири групе:

1. Регистри
2. Кеш меморија
3. Примарна меморија
4. Секундарна меморија

Регистри су уграђени у процесору, подаци које процесор обрађује, у њима су подаци док процесор их користи за израчунавања.

Кеш меморија чува податке који се често користе, може представљати бафер између примарне меморије и процесора. Ту се налазе и инструкције које се често користе или се претпоставља да ће бити коришћене. Овиме се смањује приступ спољним меморијам чиме је повећана искоришћеност процесора и самам тим систем је ефикаснији.

Примарна меморија садржи инструкције и податке које процесор тренутно обрађује, зато се она назива и радном меморијом. Посебна врста је RAM меморија - процесор може приступити њој у било ком делу RAM меморије, при чему време приступа не зависи од места на ком се налази ни од садржаја.
Друга врста ове меморије је ROM меморија, за разлику од прошле, чува податке и када је рачунар угашен. Пошто је њен назив Read Only Memory, из ње је могуће само читати податке. Чувају неке критичке програме за покретање оперативнох система. Постоје и EEPROM и јос врста које омогућавају брисање, чак и уписивање, итд.

Секундарна меморија не губи садржај након гашења рачунара, да би се њима приступило, потребно је да се учитају у радну меморију - стога је и назив *спољашња меморија.* Неки од примера су CD-ROM, хард диск, DVD, итд. Време приступа подацима зависи од њихове локације.

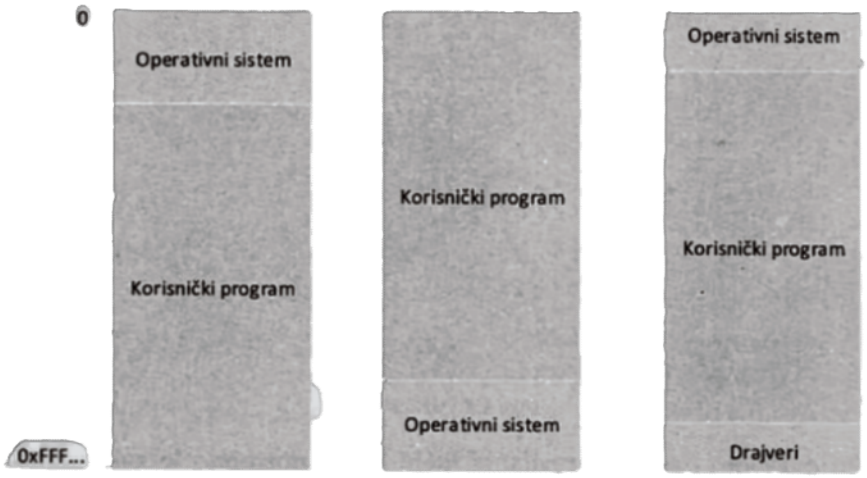
.
.
.
.
.
.
.
.
.
.

[5.1 Управљање меморијом при монопрограмирању]

5.1 Управљање меморијом при монопрограмирању

Wednesday, November 29, 2023 3:40 PM

Најједноставнији приступ управљању меморије по ком се у меморији извршава само један кориснички процес. Један део се одваја за оперативни систем, док остатак меморије користи тај једини кориснички процес који се извршава.
Постоје различите организације меморије:

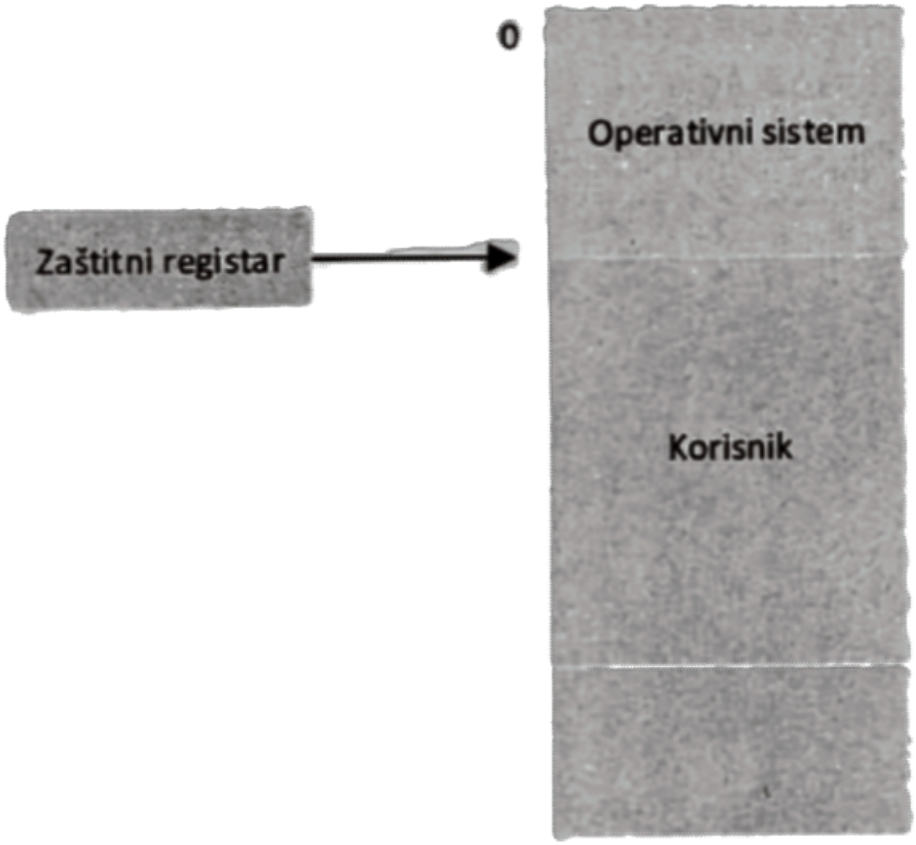


Једна подразумева да се оперативни систем учитава на ниже адресе, а процесор користи више адресе.
Друга је обрнута, процесор ниже а оперативни систем више адресе.
Трећа опција је да се део оператвиног систем учита на ниже адресе, на врху се налазе драјвери, док остатак чини кориснички програм.

Први рачунарски системи су имали овакву организацију, трећу организацију је користио MS-DOS. Функционисао је тако што корисник уноси команду за покретање програма, ОС копира захтевани програм у меморију и извршава га. Када процес заврши са радом, ОС штампа "prompt" карактер и чена унос следеће команде. За следећи унос, учитава жељено али и преписује претходни садржај меморије.

Како се диже оперативни систем?
Учитавање оперативног система у меморију се извршава приликом укључивања система.
Активира се ***bootstrap loader*** (пунилац) који преноси оперативни систем из секундарне меморије у радну меморију.
Након учитавања система, кориснички програм (процес) се пуни на адресе после оперативног система или на други крај меморије.

Када се овако управља меморијом, потребно је заштитити код и податке оперативног система од случајних/намерних измена. Овакву заштиту углавном пружа хардвер, а приступ би био такав да се свака адреса коју генерише кориснички програм пореди са садржајем ***заштитног регистра.*** Уколико је та адреса нижа од заштитног регистра, дешава се прекид и оперативни систем предузима одговарајуће акције. Углавном прекине кориснички програм и изда поруку о проблему.



Повезивање инструкција и података са меморијским адресама се може извршити у време превођења или у време пуњења.
У случају превођења, ако је заштитна адреса позната пре превођења, генерише се апсолутни код, а адресе се низају након заштитног регистра. Проблем је при промени заштитне адресе - потребно поновно превођење програма.
Када се повезује током пуњења, преводилац генерише релокатибилни код. Повезивање се одлаже до пуњења - код промене заштитне адресе, потребно је поновно пуњење програма.

У оба случаја, **заштитна адреса мора да буде фиксна** током извршавања програма.
Постоји случај када се у току извршњава програма мења величина дела меморије где је смештен

оперативни систем, чиме мењамо и заштитну адресу - овом променом долази до промене корисничке меморије на нове локације у односу на нову адресу заштитног регистра. Ова операција је јако скупа због неопходног копирања меморије.

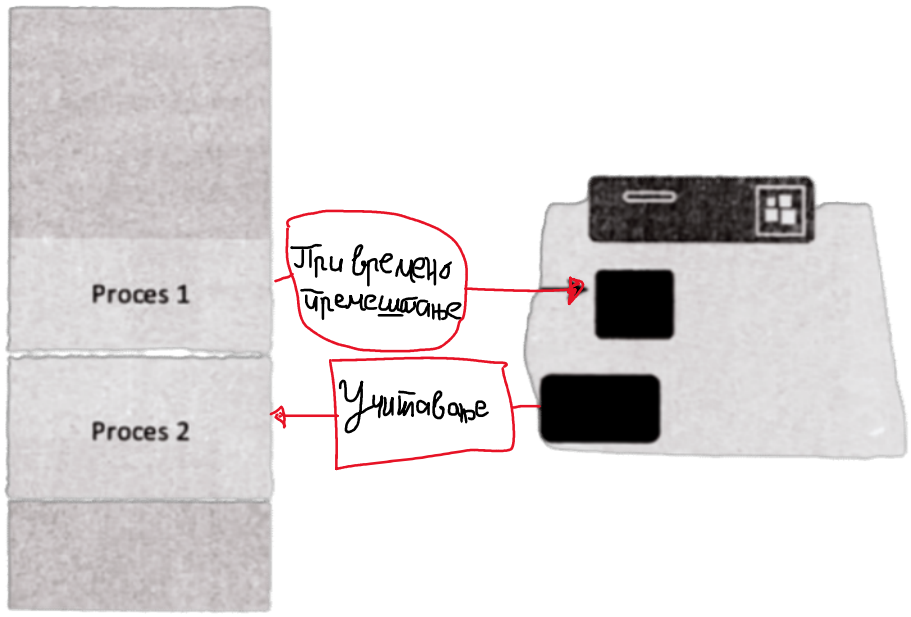
5.2 Управљање меморијом при мултипрограмирању

Thursday, November 30, 2023 4:48 PM

Постоје различите технике које обезбеђују процесима да се извршавају у систему заснованом на мултипрограмирању.

5.2.1 Пребацивање

Док се програм (процес) извршава, он мора бити учитан у меморији, али док он и даље чека на неке ресурсе - не мора да буде учитан у меморији.
Пребацивање је техника управљања меморијом (и процесима) која је погодна због добре искоришћености процесора. Идеја је да се блокирани процесор привремено премести у секундарну меморију, док се у радну меморију учита неки други процес - формира се ред привремено блокираних процеса. Из реда блокираних, формира се ред спремних процеса након што буду спремни да наставе. Када се процес привремено блокира, систем одлучује да ли ће учитати нови програм (процес) или ће активирати неки од реда спремих процеса.



За овакво пребацивање, потребна је брза и велика секундарна меморија како би прихватила меморијске слике за све процесе. Омогућити и директан приступ меморијским сликама процеса.
За време потребно за пребацивање процеса се тежи да трајање процесорских активности процеса буду сразмерно дужа од самог пребацивања због искоришћености процесора. Време се троши највише на пренос, које зависи од количине података - за бољи пренос, потребна бржа секундарна меморија.

Напреднији приступ би био преклапањем пребацивања и извршавања процеса. Док се један пребацује, други се у исто време извршава.
За реализацију овог приступа, потребно је два меморијска простора које могу прихватити две меморијске слике - два бафера су потребна.

Када процес који се извршава ослободи процесор, премешта се у слободни бафер и почиње пребацивање у секундарну меморију, док се наредни процес из другог бафера помера у област меморије предвиђене за извршавање. Затим се у одговарајући бафер учитава процес који је следећи на реду на извршавање. Овако се избегне преклапање једног процеса са учитавањем и

извршавањем других процеса у меморији.

5.2.2 Партиције

Због порасти капацитета меморије, користи се ефикаснија техника од пребацивања, а то је техника ***партиционисања меморије.*** Меморија се дели на n делова - партиција које представљају непрекидне делове меморије (партиције могу бити и једнаке и различитих величина). Свака партиција може сместити један процес - степен мултипрограмирања је n. Ако су партиције заузете, спрема се ред спремних процеса. Како се ослободи, први спреман улази на то место - ако их има више слободних, постоји начин како се одабере партиција.

Партиције треба заштитити од приступа других процеса. Хардверски се ово решава - постојањем два регистара по партицији. У зависности од имплементације, садржаће **адресе почетка** и **краја партиције** или **адресу почетка** и **величину партиције**.
Ограничен је приступ само својој партицији, иначе се предузимају одређене мере.

---слика---

Постоје два приступа партиционисању:

1. Статички - величина партиције се не мења током извршавања
2. Динамички - дозвољава промену величине, у зависности од потреба процеса и стања у систему

--слика --

5.2.3 Статичке партиције

Радна меморија се статички дели на фиксни број партиција, где свака садржи по процес. Број партиција ће зависити од: просечне величине процеса, величине радне меморије, брзине процесора,...
Величине најчешће нису једнаке - мање служе за кратке процесе, веће за захтревније.
Системи за овом имплементацијом процесе које пристужу смештају у ред спремних послова.
Планери воде рачуна о меморијским потребама процеса, као и о одређивању који процес ће коју партицију добити. Након добијања партиције, може тражити процесор за извршавање. Након извршавања, ослобађа је и неки други процес може заузети.

Две стратегије одређивања партиције:
Прва јесте да свака партиција има свој ред послова ако та партиција одговара меморијским захтевима тим процесима.
Друга јесте да се сви смештају у један ред, а да планер доноси одлуку који се процес следећи извршава. Тада се чека да се партиција одговарајуће величине ослободи. Постоји додатна модификација којом би се прескочили неки процеси како неке партиције не би биле празне - први процес чека неку партицију, постоји партиција недовољне величине за тај процес тако да у реду се прескочи тај процес који чека неку другу партицију и гледа се први следећи да се заузме та партиција која је слободна.

5.2.4 Диманичке партиције

Величине партиција нису фиксне. Системи који ово подржавају морају водити рачуна о слободној и заузетој меморији.
При покретању рачунара, одмах се меморија одваја за оперативни систем, док остатак чини једна велика слободан партиција. Како се програм упали, одваја се партиција за њега и остатак је слободан за следеће процесе након њега. Процеси како заврше са извршавањем, ослободе

заузети део партиције. Долазећи процеси траже потребну количину меморије у слободним деловима партиције. Када се ослободи меморија, ако је поред већ била слободна меморија, спајају се у тај један слободни део партиције.

Евиденцију о заутетости оперативни систем води рачуна преко bit-maps и повезаних листи. Bit-maps представља да се меморија подели на једнаке делове и да се сваком делу додели један bit - вредност 1 ако је заузет и 0 ако је слободан. Тако да за пуњење процеса се налази довољно дугачак низ нула који би био довољан меморијски за тај процес. Мапа је једноставна али претраживање празног места је дуготрајан процес, стога повезане листе су практичније.

Повезана листа се састоји од чворова при чему сваки садржи податке:

- Да ли је у том делу процес или слободна меморија
- Почетак тог дела меморије
- Дужина меморије
- Показивач на следећи чвор (наредни блок меморије)

Постоје и имплементације где су слободне и заузете меморије у различитим листама - лакше за претрагу али теже за ажурирање.

5.2.5 Фрагментација

Нежељена појава - слободна меморија коју систем "*не може*" искористити.
Фрагменција може бити ***интерна*** или ***екстерна***.

Интерна фрагментација настаје када се процес смести у партицију (простор) већу од његових меморијских потреба. Ако је партиција неке величине ***n*** при чему ми учитамо више, ту разлику чини фрагмент неискоришћене меморије. Дешава се код статичког партиционисања.

Екстерна фрагментација подразумева меморију која је слободна али не толико велика да би се у њу нешто могло сместити. Није исти проблем као код интерне фрагментације, али има свој посебан проблем екстерне фрагментације. Када би се јако дуго ивршавали процеси, десило би се да остају мали фрагменти који се не могу искористити због мањка величине меморије, али су слободни.

Проблем екстерне фрагментације се решава ***компакцијом***. Оперативни систем зауставља процесе и врши реалокације истих, тако да се слободне партиције споје, а процеси набију на неки крај меморије. Веома зехтевна операција што значи да не треба је често користити, не треба се систем заснивати на њој.

5.2.6 Алгоритми за доделу меморије

Постоје различити начини да се процесу додели слободна меморија.

Прво поклапање (First-Fit) је чест приступ. Гледа се први блок слободне меморије довољан за његове потребе - већи од његових потреба. Та меморија се означава заузетом. Време претраге меморије је мало, али је фрагментација честа.

Следеће поклапање (Next-Fit) је модификација прошлог, претрага не почиње сваког пута од почетка меморије него од тамо где је прошла стала - свакако има лошије перфомансе од прошлог алгоритма.

Случајно распоређивање је начин додељивања меморије тако да се од свих слободних и довољно великих блокова меморије изабере насумични блок за извршавање процеса. Захтевно решење јер треба одредити кандидате блокове и насумични број је потребан.

Најбоље поклапање (Best-Fit) је идеја да се процес смести у најмањи слободан блок који одговара меморијским захтевима. Захтеван је за имплементацију - пролази кроз сву меморије, проналази блокове који задовољавају потребу, израчунава међувремену величине и проналази тиме онај што је најмањи за ту потребу.

Најгоре поглапање (Worst-Fit) је идеја да се процес смести у део који се најгоре уклапа - регион са највећим капацитетом. Идеја је да се остале мање меморије искористе за мање процесе.

5.3 Страничење

Monday, December 4, 2023 6:16 PM

Један од првих начина управљању меморије који је дозвољавао да меморије додељена процесу не буде дата у једном комаду, тојест целом једном блоку меморије, него преко адреса.
Идеја је да се радна меморија подели на оквире (***frames***) фиксиране величине, а да се логичка меморија потребна процесу подели на ***странице*** које су исте величине као и оквири. Када процес треба да се учита у меморију, проналазе се слободни оквири и у њих се смештају одговарајуће странице.

За време ивршавања процеса, процесор ради са логичким адресама које садрже 2 дела:

- Броја странице - p
- Позиција у оквиру странице - d

Логичке адресе треба превести у физичке, а за то се користи структура ***табела страница.***
Оперативни систем генерише по једну табелу страница за сваки процеес који се извршава. Информације у табели јесте у којим оквирима су смештене странице. Физичка адресу добијамо додавањем вредности позиције у страници (d) на адресу оквира у којем се налази одговарајућа страница.

Величина оквира и страница је унапред дефинисана хардверски. Најчешће величина степена броја 2. Ако је величина странице ***P***, тада се на основу релативне адресе U могу израчунати број странице ***p*** и позиција у оквиру странице ***d*** на следећи начин:

- p = U div P;
- d = U mod P.

Разлог степена двојке код величине странице је да то омогућава да релативне и логичке адресе буду једнаке. Ако је величина странице $\boldsymbol{2^n}$, тада нижих ***n*** битова релативне адресе представља позицију у оквиру странице, док виши битови релативне адресе одговарају редном броју странице.
ПРИМЕР:

> Величина странице је 1КВ = 1024В = 2^{10}В и нека је 16-битном адресном простору. За релативну адресу 1327 (тј. 1327. бајту) добија се d = 1327 mod 1024 = 303 , док p = 1 --> одговарајућа физичка адреса је 1|303 (p|d).

Такође степен двојке омогућава да се пресликавање логичких у физичке адресе изведе једноставно. У табели странице пронаћи где се у физичкој меморији налази одговарајућа страница и вредност те адресе поставити на више битове релативне адресе.

> Уместо p|d
> треба f|d

За одређивање извршавања процеса, планер послова израчунава његову величину изражену у страницама и проверава расположиве меморијске оквире. За процес коме треба ***n*** страница потребно је да ***n*** страничних оквир буде слободно. Ако је слободно, странице процеса се пуне у додељене оквире, а број оквира се смешта у табелу страница тог процеса.

.
.
.

5.3.1 Хардверска подршка - табела страница

.
.
.

5.3.2 Асоцијативна меморија

.
.
.

5.4 Сегментација

Monday, December 4, 2023 11:48 PM

Преведени кориснички програм се може посматрати као скуп различитих логичких целина - сегмената. Неки од делова могу бити: главни програм, стек, хип, библиотеке, глобални подаци, итд. Идеја је да сваки од тих сегмената добије свој меморијски простор за коришћење - атомичан је, односно или ће цео сегмент бити у меморији или неће бити учитан. Сегменти **морају** бити константи, непрекидан меморијски блок, потребно је да им се додели тачно онолико колико им је потребно.

Логичке адресе код сегментације састоје се из:

1. Броја сегмената - ознака 's'
2. Позиција у сегменту - ознака 'd'

Дводимензионалне логичке адресе треба превести у једнодимензионалне - користимо табелу сегмената.

5.4.1 Табела сегмената

Сваки процес добије по једну табелу сегмената у којој се чува по један ред за сваки сегмент процеса.
Тај један ред садржи информације: редни број сегмента, адресу почетка у меморији и дужину целог сегмента. Смештена је у регистрима, асоцијативној меморији или радној меморији.
Базни регистар табеле сегмената садржи адресу локације где је смештена табела. Пошто број сегмената процеса може варирати, треба водити рачуна о величини табеле сегмената о којој води рачуна ***регистар дужине табеле сегмената.***

МАПИРАЊЕ ЛОГИЧКИХ АДРЕСА У ФИЗИЧКЕ...

5.4.2 Карактеристике сегментације

Предност у односу на остале јесте заштита меморије. Пошто је процес подељен у сегменте, неки садржи податке, инструкције. Сегменти са инструкција се могу означити да се могу само читати, без модификације чиме га штитимо од нежељених приступа, итд.

Сегментацијом добијамо могућност дељења кода и података између различитих процеса. То се омогућава ако ставке у два процеса у табели сегмената показују на исту физичку локацију.

Сегменти су најчешће различите величине, потребно је одредити меморију за сваки сегмент корисничког програма и то се постиже динамичком доделом као код страничења.

За разлику од диманичког партиционисања, овде није цео процес један блок, него само сегмент који се може наћи било где у меморији. Сегментацијом се такође решава проблем ***интерне*** фратментације, али остаје проблем ***екстерне фрагментације.*** Овде то представља мањи проблем јер је сегмент мањи од целог процеса и много се лакше уметне у неки меморијски слободан део меморије.

Комбинација сегментације и страничења се некад имплементира, како се искористе предности и умањиле појединачне мане. ***Сегментација са страничењем*** представља приступ који подразумева поделу сегмената на странице чиме се омогућава да се чак и сегменти не морају смештати у непрекидне меморијске блокове. Овако екстерну фрагментацију додатно умањујемо.

6. Виртуелна меморија

Sunday, January 28, 2024 3:27 PM

Представља начин управљањем меморијом која омогућава се процесу на располагање стави меморија која је другачије величине од оне које стварно (физички) постоји у систему. Оперативни систем је задужен да обезбеди то пресликавање.
Раније се велика радна меморија делила на мање целине, нешто на виду *time-sharing.*
У данашње време се то мисли на то да се извршава процес који није у потпуности у радном меморији. Диреткна предност је већи степен мултипрограмирања. Пошто је меморија боље искоришћена, може се више процеса учитати. Овиме кориснички процеси могу бити већи од физичке меморије. Пошто се у меморији налазе коришћени делови програма, овиме се смањује број улазно-излазних операција за замену и учитавање процеса.

Најбољи концепти вируталне меморије:

- Прекривачи
- Динамичко пуњење
- Страничење на захтев

6.1 Прекривачи

Sunday, January 28, 2024 4:20 PM

Прекривачи су засновани на идеји да се омогући извршавање програма који нису у потпуности учитани у меморију. Да се идентификују модули програма који су независни. Делови програма који се ретко користе, делови за реакција за грешку, итд, користе се у различитим временским тренуцима.

Одвајају се од главном дела програма и смештају у секундарну меморију. При покретању се динамички алоцира величина меморија за највећи прекривач, па се по потреби позива прекиравач који је потребан за даље извршавање програма. По потреби другог покривача, само ће се уписати преко већ уписаног у обезбеђеној меморији за покриваче.
Напредније верзије приступа су такве да је могуће имати више прекривача учитаних а самим тим и компликованије репосређивање и резервисање меморије.
Програмер је одговоран да сам одреди ту дељену меморију, не може се аутоматизовати.

6.2 Динамичко пуњење

Sunday, January 28, 2024 4:27 PM

Динамичко пуњење ради тако што функције и процедуре које су потребне се не учитавају меморију док не буду позване. Налазе се значи у секундарној меморији а главни је за то време у радној меморији и извршава се. При позиву функције, проверава радну, ако нема ту, налази је у секундарној.
Пунилац се позива да би обавио ово пуњење меморије, али и ажурирао одговарајуће табеле како би одговарале новом стању у систему. Након пуњења, нова функција се извршава.

Функције које нису потребне за извршавања никада ни не пуне у радну меморију, све до позива.
Не захтева подршку оперативног система.

6.3 Страничење на захтев

Sunday, January 28, 2024 4:34 PM

Најпопуларнији и најефикасније је *страничење на захтев*. Овај приступ дели програм на странице, тако да приликом покретања програма се у меморију смештају само оне странице које су непходне у првом тренутку. Смањује се искоришћеност меморије и повећава степен мултипрограмирања. Када се затражи страница, провери се да ли је у радној меморији, ако није у радној меморији, онда се иде у секундарну меморију и пребацује у радну меморију. Овиме се никада не читавају странице које нису имале потребу да буду искоришћене.

Код страничења на захтев, већина логичког адресног простора није ограничена физичком меморијом. Овиме програмер не мора да брине о капацитету расположиве физичке меморије.

Страничење на захтев се може схватити као комбинација *страничења са пребацивањем.* За разлику од класичног страничења, ова дозвољава да неке странице никад ни не буду учитане у меморију током извршавања процеса, већ само по потреби. За потребе одложеног учитавања се користи *лењи пребацивач* или *пејџер*.

Попут обичног страничења, такође се имплементира уз помоћ табеле која садржи информације о страницама. Ова има и додатну колону која чува информацију у којој се чува *бит валидности*. Страница је ,,валидна'' ако је у радном меморији, односно 1, тако и обратно. Ово је оно горе наведено да се прво провери да ли страница у радној меморији, то је провера у тој табели.
Може доћи до *промашаја странице (page fault).*

До промашаја неће доћи ако процес никада не покуша да приступи страницама које нису у радној меморији, односно ако се направи добра процена и сместе све потребне странице. Ако процес покуша да приступи страници која није у меморији, долази до промашаја а затим и до прекида. На прекид се реагује доношењем затражене странице у радну меморију.

Када се промаши, провери се да ли уопште припада том процесу. Ако не, прекида се програм и грешка при адресирању је. Ако припада, учитава се у радну меморију.
Учитава се на следећи начин:

- Проналази се слободан меморијски оквир уколико постоји и ослобађа један од заузетих уколико су сви заузети.
- Учитава се страница из секундарне у радну меморију
- Модификује се табела страница тако да бит валидност говори да је страница у радној меморији

Један од начина је *чисто страничење.* Биће промашаја али ће се учитавати само потребне скроз док се не учитају оне највише неопходне.

6.3.1 Замена странице

Један од захтева страничења јесте повећање степена мултипрограмирања, да би се што већи број проца учитавао у радну меморију. Проблеми настају када се учитава страница за коју не постоји слободан меморијски оквир.

Нека решења која се и ретку примењују су: прекине процес који тражи страницу и касније поново покрене од почетка.
Други начин је да се све странице процеса пребаце у спољашну меморију и ослободе заузети оквири, тако да када се ослободи простор онда се тек учита натраг и изврши у целости.

Најчешће решење је *замена странице*. Када је меморија пуна, а затражен је простор да се учита страница, тражи се жртва страница која ће се испразнити како би се сместила нова. Изабрана се пребацује у секундарну меморију, табела се ажурира и убацује нова.

Алгоритам би гласио:

- Наћи локацију жељене странице на диску
- Наћи слободан оквир
  - ако постоји слободан, искористи
  - ако не постоји, употреби алгоритам за одабир жртве
  - уписати жртву на диск и сагласно томе изнемити табелу страница
- учитати жељену страницу и ажурирати табелу
- наставити извршавање процеса

6.4 Алгоритми за избацивање странице

Sunday, January 28, 2024 5:23 PM

Постоји неколико алгоритама за то који ће бити описани.

6.4.1 Алгоритам заснован на случајном избацивању

Алгоритам заснован на случајном избацивању је таква да се само случајно изабере страница за избацивање. Ужасан је јер може да избаци потребну страницу и тако у круг.

6.4.2 Беладијев оптимални алгоритам

Када настане промашај странице, неке су већ учитане у меморију. Идеја је да се избаци страница која ће се најкасније искористити поново од већ учитаних, јер избацивање оне која се користи одмах након је само глупост.
Нека је редослед 1,2,3,4,1,2,5,1,2,3,4,5. Ако може максимално три учитане, за страницу 4 избацујемо 3. страницу јер се најкасније опет користи од већ учитаних.

Скоро га је немогуће имплементирати. Само је теоретски добро замишљен.

6.4.3 FIFO алгоритам

Избацивање првих учитаних страница како се покретао програм. Мана је што су те прве вероватно и есенцијалне и биће одмах избачене.
За имплементацију је потребан један меморијски ретистар по страници у који би се уписало када је одређена страница ушла у меморију, а онда би се та времена сортирала.
Најчешће се користи повезана листа у којој се смештају редно број страница. Како се учитавају, редни број се смешта на крај листе, а страница за избацивање је на почетку листе.
Тако се решава сортирање и чување времена уласка страница.

6.4.4 Алгоритам друге шансе

Модификација FIFO алгоритма, тако да се подразумева постојање једног бита по страници која се назива бит референсаности (R).
Тај бит се поставља на 1 када се страница учитава у меморији и када се приступа страници.
Потрага почиње код прве учитане странице, ако је бит 1, ставља се на 0, а време се ресетује на текуће време, тако да се страница третира као тек учитана.
Страницама се даје ,,друа шанса'' што би значило, ако се често користи, бит ће бити 1, у преводу, она која има 0, није ни скорије коришћена.
Мана ако су сви битови 1, ставе се на 0 и онда ће поново прва страница бити жртва.

6.4.5 Алгоритам сата

Модификација алгоритма друге шансе. Пошто се у алгоритму друге шансе страница премешта на крај повезане листе страница. Није извесно да ће то побољшати ефикасност.

Користи се кружно повезана листа, постоји показивач који показује на први меморијски оквир. Исто се свакој страници додели бит референсаности. Показивач иде од оквира до оквира док не наиђе на нулу и тиме се избацује. Кад нађе, убаци нову страницу и помери се за 1 да би наставио с претрагом за следећи пут.

6.4.6 LRU алгоритам

Least Recently Used се заснива да се избаци страница коришћена најдаље у прошлости.
Покушава се предвиђање у односу на захтеване странице током претходнох извршавања програма. Оне скоро коришћене ће и следећи пут бити коришћене, тако и сурпротно.

Имплементција захтева регистар који би чувао информацију о времену коришћења странице. Времена сортирати при сваком избацивању странице.
Мана је што никад не може предвидети која је следећа страница која ће бити потребна.

6.4.7 NRU алгоритам

Not Recently Used представља апроксимацију LRU алгоритма. Избацује страницу која није скоро коришћена, није нужно најдуже некоришћена. Постојање једног бита по страници је неопходно за имплементацију.

Када се страница користи, бит референсаности је 1. Пошто се може десити да се све користе, мора се десити неко периодично ресетовање битова на нулу. Треба одредити константу *t* након које се ресетују битови. После *t/2* времена након што су се битови ресетовали, неки ће бити 1, неки 0 и тада креће избацивање.

Мана, коју избацити након ресетовања зато што тад су све 0.

6.4.8 Модификација NRU алгоритма

Додаје се бит модификације (M), он је 0 када се страница учита и 1 ако се садржај странице мења. И сада могу се гледати битови (RM):

- Страница није референсана ни модификована --> 00
- Страница није референсана а јесте модификована --> 01
- Страница јесте референсана али није модификована --> 10
- Страница јесте референсана и јесте модификована --> 11

6.4.9 LFU и MFU алгоритми

Идеја *Least Frequntly Used* алгоритма је да се из радне меморије избаци најређе коришћена страница, она страница која има најмању фреквенцију коришћења. Имплементација је тежа због потребе да се редовно ажурирају те фреквенције за појединачне странице, сортирати их. Такође је и питање у ком временском периоду треба ажурирати референсање на странице.

Мана је код потребних страница које су тек ушле у систем, користиће се у будућности али у том тренутку су најређе коришћене. Тако и за странице које су се користиле често а касније и неће, остаће у меморији иако се касније можда уопште не користи.

За разлику од *LFU*, *MFU* (*Most Frequntly* Used) алгоритам је да се избацују најкоришћеније странице, са највећом фреквенцијом. Заснива се да ће странице које тек улазе бити потребније за будућност.

Очигледна мана је страница која се јако често користи, погото ако током целог програма, избациваће се често без везе.