TDDB68 sammanfattning

Martin Söderén
marso329@student.liu.se
900929-1098

March 17, 2015

1 Processer och trådar

1.1 schemaläggning

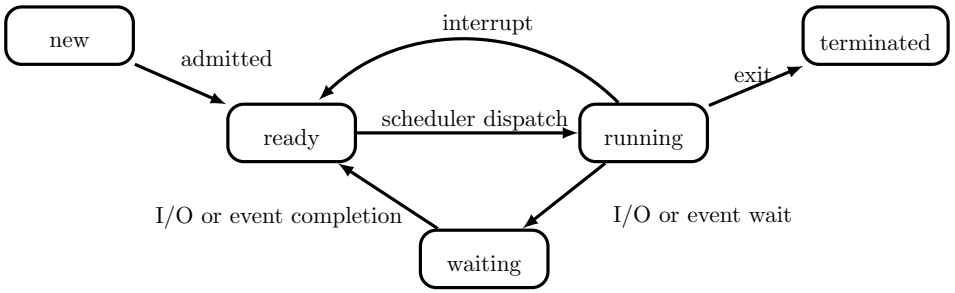


Figure 1: diagram av processstadier

1.2 Multilevel Feedback Queue

Processer placeras i en kö beroende på prioritet. När en process har körts så ökas alla andras prioritet med ett och processen placeras i kön igen.

1.3 round-robin

Processortiden delas in i tidskvoter om t.ex 100ms. En process körs då i 100ms och byts sedan ut oavsett om den är klar eller ej. Kräver preemptive.

1.4 shortest job first

Av alla processer som väntar så tas den som förvantas tas kortast tid.

1.5 FIFO

Vet du inte vad fifo är så är du körd.

1.6 preemptive schedular

Ett operativsystem som stödjer detta har möjligtheten att avbryta en process och låta en annan använda processerna istället. Motsatsen är nonpreemtive och då måste varje process självmant ge upp processorn.

1.7 process/task control block

Representerar varje process i operativsystemet. Innehåller bland annat:

- process state - new,ready ....
- program counter
- CPU registers
- CPU-scheduling information - priority and so on
- Memory management informaition - basen på stacken bland annat
- accouting information - hur mycket processen har förbrukat
- I/O status information - öppna enheter och filer

1.8 processer/trådar

En process ger resurser för att ett program ska kunna slutföras. Har ett eget adressutrymme och en egen pid. En tråd existerar inom en process och en process kan ha flera trådar.

1.9 many-to-many

I ett many-to-many system så blockeras inte alla trådar, dock alla i samma process. Det utnyttjar flera processorer. Behöver dock en schemaläggare i userland.

1.10 one-to-many

Kan skapa många trådar. Kan inte utnyttja flera cpu:er. I/O operationer kan blockera alla trådar.

1.11 one-to-one

Trådar blockerar inte varandra. Trådar kan köras på olika cpu:er. Kan bara skapa en begränsad mång trådar.

1.12 systemanrop

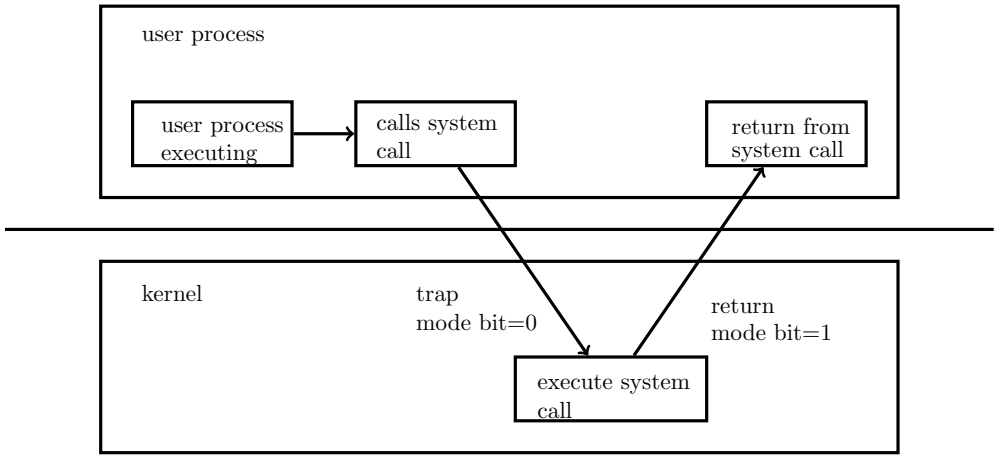


Figure 2: Flödesdiagram över ett systemanrop

2 Synkronisering

2.1 critical-section problem

En lösning måste uppfylla följande:

- Mutual exclusion - Om en process är i sin kritiska sektion kan ingen annan process vara i sin kritiska sektion
- Progress - Om några processer vill gå in i sina kritiska sektioner kan bara processer som inte ska gå in i sina kritiska sektioner avgöra vilken process som först får tillträde. Detta kan inte skjutas upp hur länge som helst
- Bounded waiting - Det finns en övre gräns på hur många gånger processer kan gå in i sin kritiska sektion efter att en process ha begärt tillträde tills dess att processen får tillträde.

2.2 reader-writer

```
semaphore mutex = 1;
semaphore db = 1;
int reader_count;

Reader()
{
  while (TRUE) {
     down(&mutex);
     reader_count = reader_count + 1;
     if (reader_count == 1){
          down(&db);}
     up(&mutex);
     read_db();
     down(&mutex);
     reader_count = reader_count - 1;
     if (reader_count == 0){
          up(&db);}
     up(&mutex);}

Writer()
{
  while (TRUE) {
     create_data();
     down(&db);
     write_db();
     up(&db);}
```

2.3 semaphores

kan implementeras med antingen avstängda avbrott eller med test-and-set. Problemet med att stänga av avbrott uppkommer i multicore arkitekturer då man måste stänga av avbrotten på samtliga kärnar. Även denna process måste vara synkroniserad. Då är test-and-set bättre då denna instruktion är atomisk och ingen process får påbörja en test-and-set innan den som utförs är klar.

Wait operationn kan utföras antingen med att processen blockas eller med ett spinlock där en while loop kollar hela tiden om låset är upplåst. Ett spinlock kan vara effektivast om låsningen är kort tid till exempel i kernel.

semaphorer är inte fifo.

2.4 monitor

En monitor är en adt som enkapsulerar funktioner och data.

2.5 *Resursallokeringsgraf*

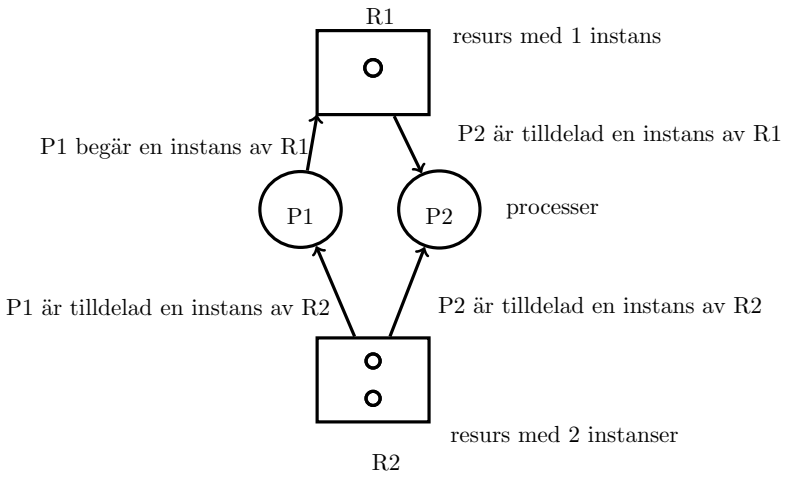


Figure 3: resursallokeringsgraf

2.6 *coffmans conditions*

- Mutual exclusion - en resurs kan bara användas av en process i taget
- hold and wait - en process håller minst en resurs och väntar på resurser som hålls av andra processer.
- no preemption - en resurs kan bara slappas frivilligt av den process som håller den.
- circular wait - en process väntar på en resurs som hålls av en process som väntar på en resurs och så vidare.

2.7 *bankers algorithm*

Varje process måste deklarera hur mycket av varje resurs den max kan behöva samt hur många den har av varje för tillfället. Systemet måste också veta hur mycket det finns a varje resurs. En process får en resurs om begäran understiger max antal totala resurser och antalet tillgängliga resurser. Annars får processen vänta. Anledningen till att denna inte används i os idag är för att man oftast inte har tillgång till hur många resurser varje process behöver.

3 meddelandesändning

3.1 *pipes*

En mängd program kedjade med hjälp av standard strömmar (stdout,stdin) så outputen från en blir inputen till ett annat program. Ett exempel är filtreringsprogram typ Linux less

3.2 *rende-vous(mötesplats)*

En process får inte fortsätta fören alla parallela processer har nått samma punkt. Vid denna punkt kan processer även överföra data mellan varandra utan mellanlagring.

3.3 *message passing*

Någonstans i minnet finns en kö som processer kan använda för att kommunicera

3.4 shared memory

Processer delar på minne som alla kan komma åt. Behöver synkronisering

4 Primärminne

4.1 data access locality

Data som hör ihop laddas oftast in i primär minnet samtidigt eller väldigt nära inpå. Viktigt i virtuell minneshantering då sidor kan ligga på olika platser i minnet men höra ihop.

4.2 clairvoyant optimal replacement strategy

byta ut den sida som kommer användas längst bort i tiden.

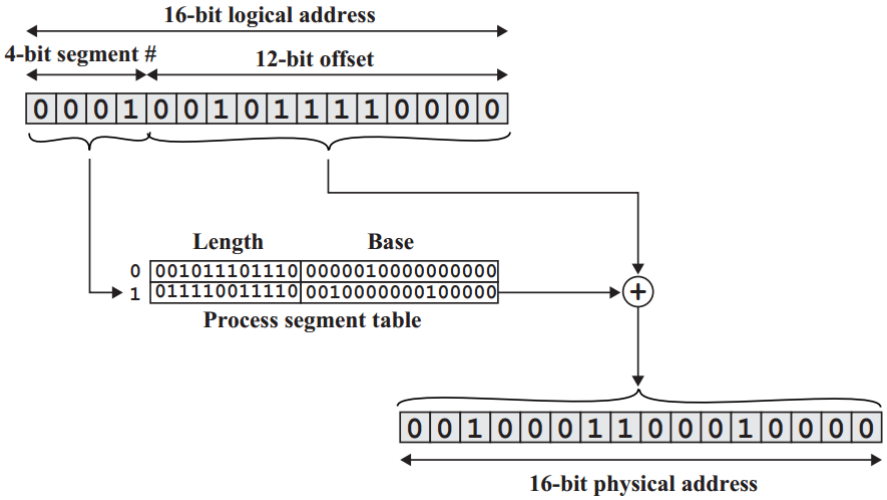
4.3 logiskt/fysiskt

Det fysiska minnet är lika stort som det logiska. Dock de fysiska adresserna används av minnesenehten medans de logiska genereras av CPU:n.

4.4 compacting

Defragmentera memory poolen. Dvs flytta ihop objekt så man får det fria utrymmet för sig.

4.5 segmentering



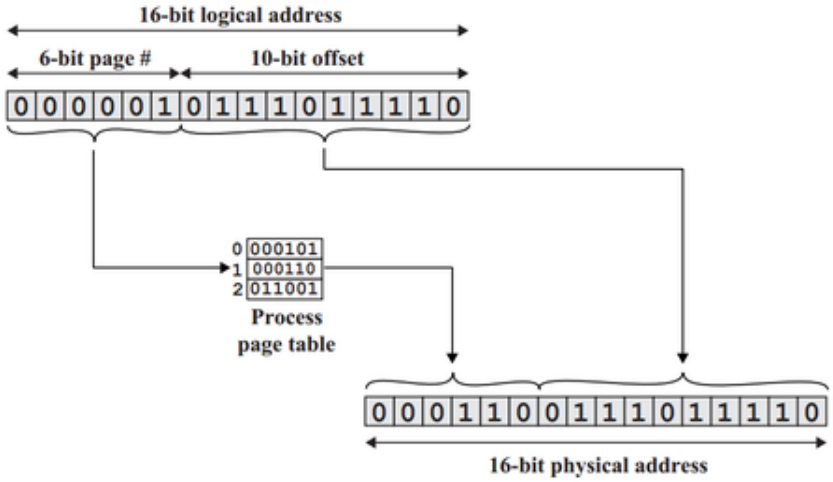
Primärminnet delas in i segment. En referens till en minnesplats innehåller vilket segment och en offset från segmentbasen. Får att komma åt minnet behövs oftast hela segmentet laddas in i minnet om man inte har paging. Segment är väldigt naturligt om man tänker på hur program är uppdelade i rutiner. Det delar också upp program och data. Utan segmentering skulle virtuellt minne nästan vara omöjligt. Segmentering är att föredra om man vill undvika intern segmentering och dela specifika datastrukturer mellan processer.

Med paging innehåller segmentet flera pages och offseten omvandlas till en page som laddas in i minnet. Ett segment kan lätt expanderas genom att det får flera pages.

4.6 MMU

Memory management unit. Omvandlar minnesrefernser till en fysiskt adress.

4.7 paging



Används när man inte kan ladda hela/alla program in i ram-minnet och måste förvara en del programmet på hårddisken. Skiljer inte på program och data. Gör det enkelt att utöka minnet då det inte kräver sekventiellt fritt minne. Pages kan delas mellan processer men då är de read-only alternativt copy-on-write, dvs vill en process skriva till en sida som delas får processen en ny likadan sida.

Om ett program försöker komma åt en sida som inte är ladded(page fault) utförs följande:

- hitta vart sidan ligger på hårddisken
- hitta en tom sida i ram-minnet
- ladda sidan till ram-minnet
- uppdatera page-table
- ge tillbaka kontrol till programmet

4.8 Thrashing

Mycket av processertiden går åt till att ladda in sidor i ram-minnet. Kan uppkomma av att en process har får få pages att arbeta med. Kan lösas med mer ram eller minska antalet program på datorn.

4.9 Translation lookaside buffer

Är ett cacheminne som omvandlar virtuella adresser till fysiska adresser. Finns den virtuella adressen in i cache så måste den söka igenom page table.

5 virtuellt minne

5.1 sidtabell

används för att omvandla den virtuella adressen till en fysiskt. Görs för varje minnesreferens så det måste gå snabbt.

5.2 Enhanced FIFO second-chance

Alla sidor lagras i en ringbuffer och har en use-bit. Use-biten 1-ställs när sidan först laddas och varje gång den refereras till. Varje gång man söker igenom buffern bakifrån så 0-sätts alla use-bit. Den första med en use-bit=0 tas bort.

5.3 Write-through

CPU:n använder ett cache minne för att skriva till långsammare minnen. Skrivs direkt och synkront.

5.4 delayed-write

Datan lagras ett tag i ett temporärt minne och skrivs senare till det långsammare minnet. Bra där data används temporärt av ett program och raderas nästan direkt.

6 Filsystem och sekundärminne

6.1 file control block

Heter inode i Linux. En kopia på alla öppna fcb:s ligger i kernel minne och innehåller vad användare får göra. När den har blivit ändrad., ägare, storlek och pekare till blocken.

6.2 soft/hard links

mjuka länkar innehåller hela filnamnet så om namnet på en fil ändras så blir länken ogiltig. En hård länk innehåller en pekare till datan så om datan flyttas så är länken ogiltig.

6.3 virtual file system

ett abstraktionslager mellan os:et och hårdvaran. Detta gör det möjligt att os:et kan köras på flera olika filsystem.

6.4 cache

En hårddisk har ett cacheminne som innehåller de senaste inläsningarna. Detta gör att average read time är lägre än actual read time. När os:et begär data så kan den begära mer än vad det behöver så att datan ligger i cache minnet, till exempel om det håller på att läsa in en lång fil.

6.5 internal fragmentation

om ett program begär 23bytes så får det 32 istället för minnet är uppdelat så.

6.6 external fragmentation

filer sprids på olika delar av hårddisken så det tar lång tid att läsa in hela filen.

6.7 länkad allokering

Varje fil är en länkad lista med diskblock. Katalogen vat första och sista pekaren. Varje block har en pekare till nästa block.

6.8 sammanhängande allokering

Varje fil ligger på ett samlat ställe på disken. Katalogen vet startpekaren och storleken.

6.9 indexerad allokering

Alla filer har ett indexblock som innehåller alla pekare till filens block. Katalogen vet files indexblock. Det blir ett overhead för att spara alla pekare.

6.10 FAT

som indexerad fast det finns en tabell över alla pekare mellan blocken.

6.11 inode

Representerar en fil på en hårddisk i operativsystemet. Innehåller bland annat(POSIX):

- storleken på filen
- på vilken enhet den ligger
- användar id
- group id

6.12 utsvältning

En del schemaläggningsalgoritmer kan göra så att visa processer aldrig får tillgång till CPU:n. Till exempel om man använder shortest-job first och bara tittar på hur lång tid en process tar. Då kan en process som tar lång tid bli utsvulten. För att motverka detta får man ta hänsyn till hur länge en process har väntat.

6.13 first come first serve

den som först begär hårddisken får först tillgång till den

6.14 shortest seek time first

Sökningen som ligger närmast hårddisken nuvarande position kommer först.

6.15 Scan algorithm

Hårddisken åker fram och tillbaka över disken och när den är nära en sektor som någon vill åt hämtar den datan.

6.16 c-scan algorithm

som scan fast den gör inget på tillbakavägen

6.17 förbättre prestande

Raid 0

7 skydd och säkerhet

7.1 skydd

mekanism som kan användas för att kontrollera tillgång till resurser.

7.2 säkerhet

7.3 setuid

Om en fil har setuid satt så körs den filen som om den vore startad av ägaren. Om en fil har setuid satt och ägs av root så kan detta vara en säkerhetsrisk.

Skydda innehållet från yttre medveten påverkan.

7.4 dual-mode operations

Instruktioner som ändrar MMU-register och liknande kan inte utföras från user level programs. Endast från en interupthandler i kernel mode.

7.5 capability list implementation

varje användare har en lista över vad den får göra med dokument.

7.6 buffer-overflow attack

När ett program medveten överskrider en buffers storlek för att skriva över data. Detta kan göra det möjligt att attakera program och starta egna program som har samma privilegier som det attackerade programmet. Även läsa data som inte tillhör processen(heartbleed). Man kan motverka detta genom att göra boundchecks på systemanrop.

7.7 salt hashing

gör rainbow tables omöjliga.