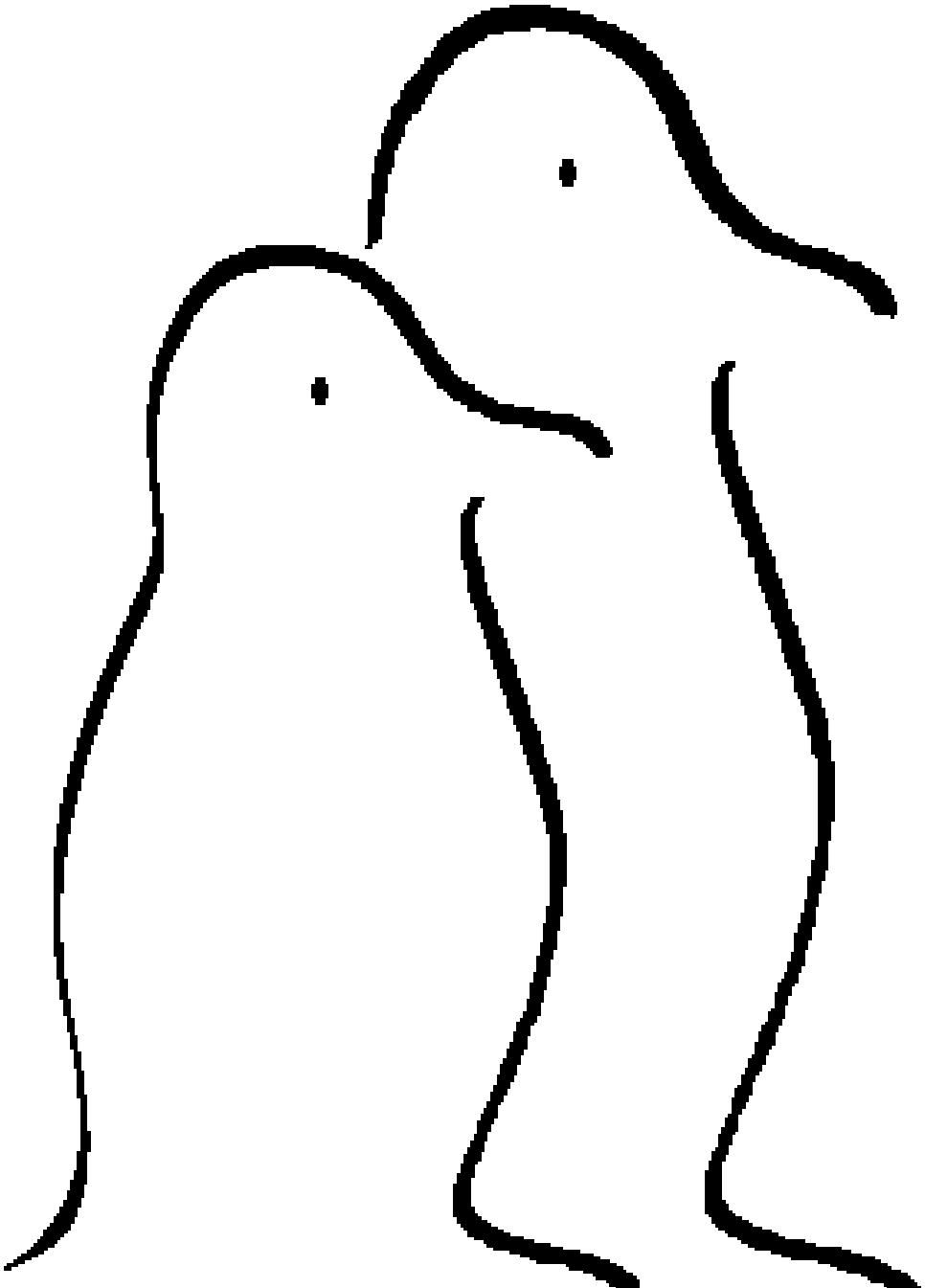


Linux Skeduleren

Af

Kim M. Jensen & Gregers Nakman
Datanom, Operativsystemer 1998

Copyrights

MSDOS (DOS), Microsoft, Windows 95/98 og Windows NT er registrerede varemærker ejet af Microsoft Corporation.
Open Source er et registreret varemærke ejet af Eric S. Raymond på vegne af den frie softwareverden
UNIX er et registreret varemærke ejet af The Open Group.
Linux er et registreret varemærke ejet af Linus Torvalds.

Resume

Følgende rapport er udfærdiget november 1998 som temaopgave på modulet Operativsystemer under datanomstudiet på Niels Brock copenhagen business school.

Vi vil i detaljer gennemgå de metoder, der benyttes til Linux Red Hat 5.1 skeduleren. Vi har fokuseret på 3 områder:

- Datastrukturer
- Processkifte
- Prioritering

Det vil fremgå, at skeduleren ikke er effektiv, når antallet af aktive processer er stort. Ligeledes vil det blive sandsynlig, gjort at real-time processer prioriteres forkert.

Vi har i øvrigt under vores gennemgang konstateret at skeduleren har 1 primær og 2 sekundære skedulerings-algoritmer, hvilket har givet anledning til en del forundring. Hvorfor ??.

Indledning

Microsoft er nutidens alt-dominerende faktor på operativsystemmarkedet, på trods af visse indlysende mangler. I dag er der kun 3 – 4 væsentlige typer af operativsystemer på markedet. De største typer er VMS (mainframes), UNIX, AS/400, Windows. Windows er idag det mest brugte system indenfor små og mellemstore virksomheder, mens UNIX og AS/400 stadig har den største del af de ”store” systemer. Dette hænger meget tæt sammen med begrebet stabilitet. Her har Microsoft Windows igennem tiden vist store svagheder, og man spøger da også ofte med, at der sjældent i historien er blevet solgt en vare med så mange fejl i så mange eksemplarer. Det er derfor meget interessant at kigge nærmere på et af de operativsystemer, der giver Microsoft noget udfordring på deres operativsystem-marked. Der er i øjeblikket meget omtale af to forskellige systemer, hvor det ene (BeOs) er rettet mod grafiske/multimedia miljøer, medens det andet kaldet Linux er rettet mod udviklings- og transaktions-orienterede miljøer.

Vi har valgt at gøre rede for skeduleren, da denne er et af de mest centrale emner indenfor ethvert multitasking operativsystem. Linux er et operativsystem, der er registreret under Open Source (kort

forklaring på Open Source følger), hvilket gør, at al dokumentation og source-kode er frit tilgængeligt. Derfor kan vi komme meget mere i dybden af dette system, end vi kan med noget andet kommercielt operativsystem.

Vi finder det interessant at betragte de processer og strukturer, der er i forbindelse med skeduleren, samt at kigge på om den skeduler, der findes, kan gøres bedre, hvilket vi regner med at den kan. Vi vil prøve at belyse, hvorledes skeduleren kommunikerer med processer, og i givet fald hvornår og hvorfor. Endvidere ønsker vi at finde svar på, om skedulerens processer og strukturer er optimale i forhold til det stykke arbejde, der skal udføres, samt i forhold til real-time skedulering. Det er vores fornemmelse, at real-time blot er taget med af hensyn til fremtiden, men at den reelt ikke understøtter dette.

Det er vigtigt at understrege, at dette ikke vil blive en sammenligning af de forskellige operativsystemer, men en isoleret gennemgang af funktionerne i Linux skeduleren for Red Hat distributionen v. 5.1.

For hvert afsnit vil vi afslutningsvis angive kilder for pågældende afsnit angivet i firkantparanteser ([ ]). Kilder er at finde bagest i rapportens litteraturliste.

Verden omkring Linux

Linux historie

5. oktober 1991 annoncerede Linus Torvalds på *comp.os.minix* nyhedsgruppen, at version 0.02 som den første officielle version af linux. På dette tidspunkt kunne Linux ikke meget andet end at afvikle en *bash* (GNU Bourne again shell) samt afvikle *gcc* (GNU c-compiler). Linux blev oprindeligt startet som et hobbyprojekt for Linus, mens han var studerende på det finske Universitetet i Helsinki, Finland, og var inspireret af Minix, som var udviklet af Andrew S. Tanenbaum. Flere og flere personer gav deres, for at udvikle Linux med fokus på kernefunktioner. I december 1993 var der dog stadig ikke kommet en kerne med version 1.0, men man var dog kommet så langt at X-Windows kørte. Linux har siden 1993 udviklet sig voldsomt, meget i takt med udbredelsen af internettet, og eftersom at Linus har gjort Linux frit tilgængeligt under Open Source, er dette system blevet mere populært. I dag nærmer kernen sig version 2.2 med både support for symmetrisk multiprocessering og plug and play, samt læsning af ”alle” kendte diskformater (FAT32, NTFS, HPFS, ISO9XXX).

Open Source

Man har i mange år haft begreberne ”free software”,”freeware” og ”public domain”, men betegnelserne har været udsat for megen diskussion. Specielt i engelsktalende lande har der været et problem, idet at ”free” både kan betyde ”Fri” og ”Gratis”. Man har derfor opfundet konceptet Open Source, der er en fuldkommen fast defineret standard, hvilket gør, at der ikke skulle kunne opstå misforståelser. Efter at have lavet en klar definition af hele konceptet er Open Source blevet registreret som et beskyttet navn

og dermed kan grundlaget ikke anfægtes. Alle kan anvende konceptet Open Source uden afgifter, men under den betingelse at konceptet anvendes korrekt.
At et produkt bliver placeret under Open Source betyder, at alt omkring det pågældende produkt bliver frit tilgængeligt, hvilket vil sige, at både kildetekst og dokumentation er offentligt tilgængeligt. Gode eksempler på produkter, der findes under Open Source, er netop Linux, samt Netscape Navigator. Folkene bag Netscape har ligefrem benyttet Open Source som et våben i krigen mod Microsoft, hvilket giver Netscape en kæmpe fordel. Ved at placere Netscape Navigator under Open Source har Netscape fået et 10 fold af programmører, der hver især og i diverse projekter forbedrer, tester og dokumenterer udviklingen af Netscape. På trods af de gratis produkter, er det stadig muligt at lave en ganske fornuftig forretning. Eksempelvis tjener Netscape gode penge på deres internetservere, og andre produkter, som er meget tæt knyttet til Netscape Navigator. Andre firmaer tjener penge på at distribuere og endnu andre på at servicere.

Symmetric Multi-Processing (SMP)

Tilstedeværelsen af flere cpu'er gør tilværelsen for skeduleren mere kompliceret, da den skal håndtere balanceringen af arbejde mellem processorene. I et single processor system er den idle proces den første proces i task-vektoren. I et SMP system er der en ideel proces pr. CPU, og man kan have mere end en idle CPU. Yderligere skal man ligeledes tage højde for, at der er en aktiv proces pr. CPU. SMP systemet må derfor holde styr på alle aktive og idle processer for hver CPU. Alle processer kan køre på alle CPU'er (dette kan dog begrænses vha. *processor*-feltet i *task_struct*, samt en så kaldt *processor_mask*), men skeduleren vil dog give en lille fordel til processen for at køre på den samme CPU flere fortløbende gange.

Kilder [2, 5, 9, 10]

Linux Skedulere

Skedulerens datastrukturer

Alle processer er repræsenteret vha. en datastruktur kaldet *task_struct*. I den del af hukommelsen, der er reserveret systemet, har linux en såkaldt *task_vector*, som er en tabel af pointere til processer af typen *task_struct*. Denne *task_vektor* er en tabel med pointere til alle *task_struct* i systemet. Mængden af processer er dermed begrænset til mængden af pladser i *task-vektor* tabellen (default 512). En af disse processer er den statisk deklarerede *init_task*, der bl.a sørger for at starte dæmonerne op. *Init_task* er den første proces i *task_vektor*. *Task_vektor's current* pointer peger på den proces, der i øjeblikket er tildelt en timeslice (200 ms).

Her er en beskrivelse af den *task_struct*, der oprettes for hver proces, som skal afvikles under linux. Alle felterne er ikke søgt beskrevet - der er over 50 - men de som er essentielle for forståelsen af, hvorledes skeduleren arbejder. Den interesserede læser vil kunne finde den fuldstændige deklaration af *task_struct* i appendix A.

"Run-køen": Alle processer er udstyret med en pointer (*next_run*) til den næste process, der er klar til at køre, samt en pointer (*prev_run*) til forrige (STATE==TASK_RUNNING). Alle processer findes i den før omtalte *task_vector*, men kun de processer, der er klar til at få CPU-Tid, vil have pege at på noget med *next_run* og *prev_run*. Med disse pointere bliver der således opbygget en dobbelt-hægtet – liste.

STATE: En proces kan være i en af 6 tilstande, hvoraf de vigtigste er nævnt i det følgende (alle definitionerne kan ses i appendix A):

- TASK_RUNNING vil sige at processen er klar til at få tildelt en timeslice
- TASK_INTERRUPTIBLE, dvs. processen er blokeret (sover) og venter på et signal, som den kan behandle, eller TASK_UNINTERRUPTIBLE, hvor processen er blokeret i venten på en ressource.
- TASK_STOPPED benyttes når processen har modtaget signal fra en anden process - eks. en debugger - om at stoppe
- TASK_ZOMBIE vil sige at processen stadig har en *task-struct* i task_vectoren, men foretager sig tilsyneladende ikke andet end at spilde skeduleringsrutinens tid.

SIGNAL & BLOCKED: Disse to felter benyttes til at give skeduleren information om, hvilken tilstand processen befinder sig i og hvorfor. Ved hjælp af *signal* kan man kommunikere med processen for eksempelvis at vække processen efter et kald til noget IO.

IDENTIFIER: Hver process er tildelt et unikt procesnummer (*pid)*. Dette *pid* bliver givet processen når der udføres et *fork()* systemkald. Forældreprocesser kan da holde styr på deres børn, da forælderen kan benytte *pid* til at teste barnet. Desuden er der et felt til angivelse af brugerne og tilhørende grupper

(*uid* & *gid*), så det kan kontrolleres, at processen ikke overtræder sine rettigheder i forbindelse med eksempelvis filtilgang.

LINKS: Vigtigste pointer i forbindelse med skeduleringen er den pointer til andre processer i TASK_RUNNING (*next_run* eller *prev_run*. Det er en dobbelthægtet liste). Desuden er der bl.a. pointere fra en proces og til andre processer, som de opstår under fork-systemkaldet. Således er det muligt at bestemme den enkelte proses' forældreproces, egne søskende og børneprocesser.

TIMERS:Processens *task_struct* indeholder felter, bl.a. *counter* til angivelse af processens tidsforbrug, som igen benyttes af skeduleren til prioritering. Dette forbrug opgøres i såkaldte jiffies, som er antallet af systemets clock-ticks (10 ms) siden processen blev oprettet og angiver tidsforbrug opdelt i User og System mode. Desuden er det muligt for processen at "bestille vækning" til bestemte tidspunkter og således bruge disse tidsintervaller som beslutningsgrundlag for dens videre programafvikling.

CPU SPECIFIC CONTEXT: Den enkelte proces' omgivelser skal gemmes ved processkifte. Disse omgivelser består af CPU-registre og processtak, programcounter og flag, som de stod efter processens sidste programlinieudførelse.

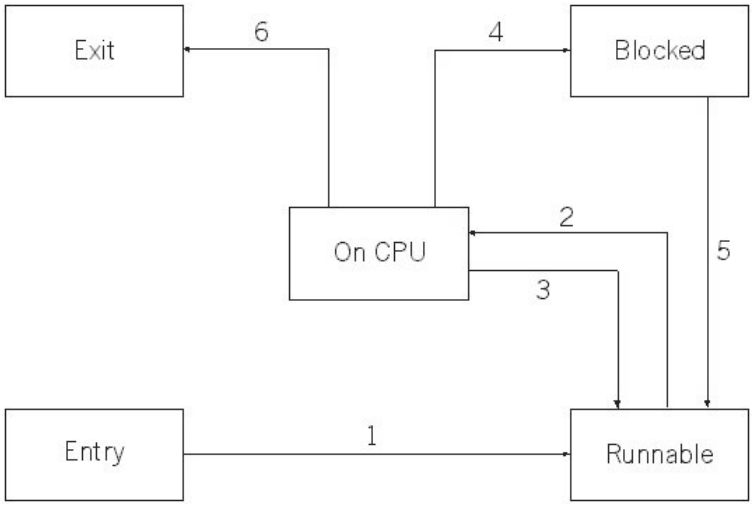
Delkonklusion

Det kan synes noget begrænsende, at der kun kan køre 512 processer af gangen. Umiddelbart ville en løsning på dette problem være at definere en kerne-parameter til angivelse af tabellens størrelse således, at den gav plads til for eksempel 2048 processer.
Yderligere kunne man forestille sig en ”run-kø” sorteret efter *counter*. Dette ville imidlertid medføre et systemoverhead til administration af den sorterede liste.

Kilder[3, 4, 10]

Processkifte

Linux operativsystemet er preemptivt. Således kan en proces ikke selv bestemme, hvor lang tid den vil optage CPU'en. Processerne gennemgår som tidligere nævnt flere STATES i løbet af deres levetid. Disse kan grafisk eksemplificeres med følgende transitionsdiagram:



Figur 1 Transitionsdiagram

1. Processen startes (med *fork ()* se appendix D) og tildeles en indgang i task_vektoren. *Counter* i processens *task_struct* tildeles en værdi baseret på processens prioritet (fra feltet *priority*).
2. *Schedule()* vil nu tage processen i betragtning, hvad angår CPU-tid. Den proces, som er i STATE TASK_RUNNING og har højest counterværdi vil blive allokeret CPU-tid.
3. Når processen har opbrugt sin timeslice vil den blive lagt tilbage i køen af processer i STATE TASK_RUNNING.
4. Hvis en proces forespørger ressourcer (I/O) eller foretager andre blokende systemkald (eks. semafor-operationer), så vil processen blive overført af kernen til kø for pågældende ressource og *schedule ()* vil bestemme hvilken proces, der herefter skal tildeles CPU-tid. Processens STATE bilver af kernen sat til enten TASK_INTERRUPTIBLE eller TASK_UNINTERRUPTIBLE.
5. Når en ressource bliver klar til en proces, vil processens STATE blive sat til TASK_RUNNING. Hvis det er et såkaldt interrupt vil processen umiddelbart blive tildelt tid på CPU'en. Men under normale omstændigheder, overgår processen bare til køen af processer i STATE TASK_RUNNING og herefter betragter skeduleren alle disses *counter*-værdi for at bestemme hvilken process, der nu står for tur.
6. En proces kan afslutte sin programafvikling, idet den når til slut på program, eller den kan blive "smidt af" af andre processer.

Skeduleringen foregår som en del af kernen. Det er en funktion, som kaldes hver gang en proces foretager en overgang i ovenstående diagram, da det er nødvendigt herefter at foretage bestemmelse af

hvilken proces, der under de nye omstændigheder, har forbrugt mindst CPU-tid i forhold til de enkelte prosessers oprindelige prioritet.

Her følger en beskrivelse af de relevante dele af kernefunktionen *schedule ()* i pseudo-kode. Indledningsvis gennemløber skeduleren sin task queue (se evt. appendix A for kø-strukturen) for at udføre jobs, som er bestilt af andre dele af kernen eller device drivers. Dette består bl.a i at inkludere processer, der har modtaget signal om tilstandsskift til TASK_RUNNING, idet de nu kan afvikles. Alle disse processer vil nu være med til at danne grundlag for en ny vurdering af hvilken proces, som vil blive næste *current.*

Som omtalt vil skeduleren blive kaldt, idet et processkifte skal finde sted. Lad os betragte en situation, hvor en proces er den nuværende aktive proces i CPU'en og foretager et systemkald, der resulterer i et hændelsesforløb, hvor skeduleren aktiveres. Vi forestiller os en kørende proces, som gerne vil tilgå data fra en allerede åbnet fil. Følgende er en overfladisk gennemgang, og vi vil ikke komme ind på

```
Udfør de funktioner som er i skedulerens task kø (tq_struct).

FOR nuværende aktive process DO
        IF nuværende aktive process er Round Robin (RR) THEN
                Udfør RR Vedligeholdelse
        End IF

        IF processens har en real time interval timer kørende
                IF interval timer er udløbet
                        Send SIGALARM til proces
                        Genstart interval timer, hvis nødvendigt
                END_IF
        END_IF

        IF processens STATE er TASK_INTERRUPTIBLE
                IF processen har modtaget signal eller dens timer er udløbet
                        Sæt processens STATE til TASK_RUNNING
                END_IF
        END_IF
END_FOR

FOR hver proces DO
        IF processens STATE er TASK_RUNNING
                Husk den proces med højest counter, restende tid tilgode
        END_IF
END_FOR

IF alle processer har forbrugt deres CPU-tid
        FOR hver proces DO
                Beregn processens nye counter værdi udfra dens oprindelige prioritet
        END_FOR
END_IF

Foretag processkift til den proces med højest counter værdi
```

Figur 2 Skeduler pseudo kode

driverens oprettelse af "requests" for bufferdata, ligesom controlleren for enheden heller ikke nævnes.

A. Brugerprocesen er i STATE TASK_RUNNING, og er den nuværende aktive proces. Processen vil indlæse data fra fil på harddisk.
B. Dette systemkald, en C-biblioteks rutine, loader CPU'ens registre med identificerende parametre, hvorefter det generiske Int Ox80 trap udføres. Dette fremtvinger skift fra Usermode til Kernel mode, således at kernen overtager kontrollen.
C. Kernen ændrer brugerprocessens STATE til TASK_UNINTERRUPTIBLE og identificerer via dispatcher rutine, hvilket systemkald, der ønskes udført. Pågældende device driver's interrupt rutines *read ()* funktion startes under brugerprocessens omgivelser, hvilket letter kopieringen af data fra cash buffer i kernens hukommelsen til brugerprocessens databuffer.
D. Device driver kalder nu kernefunktionen *sleep_on ()*, hvor argumentet er en pointer til strukturen Wait_queue, som er dedikeret den enkelte ressource. Vi har således her en driver, som arbejder i kernel mode på vegne af brugerprocessen.
E. Skeduleren overtager nu, da det skal vurderes, hvilken proces, som herefter mest fortjener at få tildelt CPU-tid.
F. I/O enheden afgiver efter et tidrum interrupt, som angiver, at driveren nu kan overføre data fra enheden til cash buffer, hvorefter kernekaldet *wake_up()* kaldes med argument, som angiver hvilken wait_queue, der skal aktiveres. Dette element indsættes i skedulerens Task Queue (*tq_struct*), og er altså en pointer til en kernefunktion, som ønskes udført næste gang skeduleren startes.

Delkonklusion

Det er her interessant at iagttage, at samspillet mellem driver, kerne-kald og skeduleren kan foregå på to forskellige måder. En driver kan eksempelvis have noget data klar til en given proces, men denne proces kan sagtens vente med at blive startet. Derfor vil driveren blot lagre denne information i proceskøen, som vil blive afviklet næste gang *scheduler ()* bliver kaldt. Under andre omstændigheder vil driveren lægge informationen i proceskøen, og umiddelbart efter lave et kald til skeduleren. Dette vil bevirke at informationen bliver overført til processen hurtigst muligt.

Kilder [3, 10]

Prioritetsstyring

Prioritering håndteres af skeduleren ved at benytte information om hver enkelt proces, som er gemt i *task_struct*,. Informationen som skeduleren benytter sig af gemmer sig i fire forskellige variable: *counter, priority, policy* og *rt_priority*. Feltet *policy* kan antage værdierne 0, 1 og 2, der hhv. står for SCHED_OTHER, SCHED_FIFO og SCHED_RR. Denne rækkefølge er tilfældig, da Linux skeduleren er en prioriteret skeduler med Round Robin (RR) og FIFO som sekundære algoritmer. Dette betyder, at det først og fremmest er prioriteten der bestemmer hvilken process, der skal have CPU-TID, mens den sekundære skeduleringsalgoritme bestemmer rækkefølgen af de processer, der skal have CPU-Tid, i tilfælde af lige høj prioritet. SCHED_FIFO kan benyttes til at angive, at et sæt af processer skal håndteres som FIFO, men ved nærmere analyse af skedulerens kildekode (se appendix B) se det, at der kun er tale om en pseudo fifo.

FIFO

Ved FIFO har skeduleren allermindst at lave. Faktisk ingenting hvad angår prioritering af processer. Hvis det antages, at alle processer har lige høj prioritet, og denne ikke ændres under procesafvikling, så vil den proces, der først kom i "run-køen", først blive afviklet.

Round Robin (RR)

RR håndteres af 4 linjer kode:

```
if (!prev->counter && prev->policy == SCHED_RR) {
        prev->counter = prev->priority;
        move_last_runqueue(prev);
}
```

Figur 3 RR Håndtering

Metoden er faktisk ret simpel. For at undgå at prioriteringen (se senere) kommer til at styre processen og ikke RR gør man det at såfremt *counter* ikke er 0 resetter man counter-feltet, og flytter processen ud som den sidste proces i "run-køen". Hvis man antager at programmøren af programmet tildeler alle processer samme prioritet i feltet *priority*, så vil placeringen i "run-køen" være afgørende for, hvornår processen vil blive betjent. Dette vil blive uddybet i afsnittet omkring prioriteringsskedulering.

Prioriteret skedulering

Alle processer vælges udfra, hvilken prioritet processen har. Dette må ikke forveksles med feltet *priority* i *task_struct*, da dette er den prioritet som processen er født med, men ikke nødvendigvis den nuværende prioritet. For hver gang processen får CPU-tid, vil feltet *counter* i *task_struct* blive talt ned med antallet af jiffies (cpu-tids-enheder), der forløb under procesafviklingen. Linux sammenligner *counter* i processernes kapløb om at få CPU-Tid, og ikke *priority*.

Feltet *rt_priority* er feltet over dem alle, når det kommer sammen med en proces, der har *policy*==SCHED_OTHER. Dette felt er forbeholdt real-time processer, der i modsætning til alle øvrige processer, af natur sætter krav om meget højt og hurtigt throughput. Dvs. at en real-time proces skal betjenes meget hurtigt. Real-time processer vil blive betjent på FIFO maner, mens prioriterede processer vil få talt deres *counter* ned med det antal *jiffies* processen forbruger, mens den er aktiv i CPU'en. En periode i CPU'en vil altså nedskrive processens prioritet.

Proces-udvælgelse

Som tidligere nævnt er processen født med en prioriteringsværdi i feltet *priority*, og dette er en statisk værdi, men den kan ændres ved hjælp af systemkaldene *nice ()* og *setpriority (). Nice ()* er en kommando som alle har lov til at benytte til at sætte sin statiske prioritet ned med, hvorimod *setpriority ()* kan ændre prioriteten i begge retninger. *Setpriority ()* kan derfor kun benyttes af root.
Når skeduleren startes (med systemkaldet *schedule ()*), vil den først undersøge om den nuværende aktive process har *policy* = SCHED_RR. Er dette tilfældet, fortages ovennævnte behandling af RR-processen. Denne handling gør SCHED_RR til sekundær algoritme i forhold til SCHED_FIFO.
Skeduleren vil derefter gå igang med selve kapløbet om cpu-tiden. Dette kapløb bliver håndteret indenfor 6 liner c-kode:

```
while (p != &init_task) {
        int weight = goodness (p, prev, this_cpu);
        if (weight > c)
                c = weight, next = p;
        p = p->next_run
}
```

Figur 4 Proces-udvælgelse

p er en pointer til den *task_struct* der bliver testet, **prev** er en pointer til den nuværende aktive process' *task_struct*. **this_cpu** er uinteressant, da den kun har relation til SMP.
Det er funtionen *goodness ()*, der står for at hente *counter* ud af *task_struct*. Af hensyn til SMP, har man valgt at lave *goodness ()*, i stedet for blot at inkludere nedenstående kodelinier direkte i skeduler-koden. Ses bort fra SMP-delen er *goodness ()* meget enkel:

```
static inline int goodness (struct task_struct *p, struct task_struct *prev, int this_cpu)
{
        int weight;

        if (p -> policy != SCHED_OTHER)
                return 1000 + p->rt_priority;
        weight = p->counter
        if (weight )
                if (p == prev)
                        weight += 1;
        return weight.
}
```

Figur 5 Goodness funktionen

Desværre ser det ud til, at der her er en fejl i koden til skeduleren. Den første if-sætning burde have været "if -> policy == SCHED_OTHER)". Meningen med denne if sætning er at give real-time processer en stor fordel i kapløbet. Alle real-time processer skulle blive forsynet med ikke mindre en 1000 jiffies + deres realtime prioritet (*rt_priority*). Desværre ser det ud som om, at det er alle andre processer end realtime processer der får de 1000 jiffies + hvad der måtte stå i p->*rt_priority*.

Hvis vi antager at *goodness()* fungerer korrekt, vil en real-time proces blive væsentligt forfordelt, da der hver gang vil blive retuneret et tal større end 1000. Er processens *policy* forskellig fra SCHED_OTHER, vil man tage indholdet af processens *counter*, og hvis den proces vi tester på samtidig er den proces, der er den nuværende aktive proces, vil skeduleren give den en lille fordel ved at lægge 1 til dens *counter*. Dette er en meget fornuftig disposition, da den nuværende proces allerede har hentet diverse informationer ind i hukommelsen. På trods af indgående studie af både definitioner og *fork ()* (se evt. appendix D) har det ikke været muligt at fastslå, hvad maximum og minimum værdierne er for *priority*. Det må dog antages at maksimum værdien ikke overstiger 1000, idet det ville bryde med princippet om real-time- processer.

Som det ses af fig. 4 vil skeduleren løbe alle processer igennem, som er at finde i "run-køen", og til slut vil pointeren *next* pege på den proces, der har den største prioritet.

Delkonklusion

Kort opsummeret fungerer prioriteringen i Linux skedulerens udvælgelses-mekanisme ved at sammenligne indholdet "en vægt", som for real-time processers vedkommende er 1000 + *rt_priority*, og for alle andre processer er det indholdet af *counter*. Ved gennemløb af "run-køen" vil den process, der først har den højeste vægt blive valgt som næste proces i CPU'en.

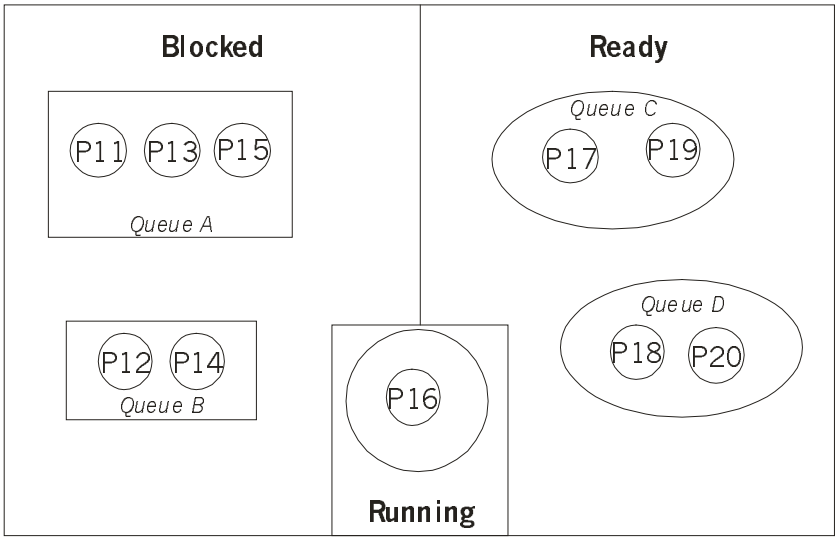
Man kunne forestille sig en simpel forbedring af skeduleren, hvor en enkelt kø håndtere alle real-time processer, og denne kø er det første, der bliver checket før end noget andet. Dette koster selvfølgelig noget i administration af køen, men det vil blot være noget pointer-håndtering, hvilket er meget hurtigt. Man vil på denne måde få afviklet alle real-time processer meget hurtigt, uden at skulle løbe hele "run-køen" igennem og så opdage, at der er en real-time proces. Faren ved en dedikeret kø, er at almindelige processer kan blive udsultet, hvis mange real-time-processer skal afvikles gentagne gange med korte tidsintervaller. Man kunne spørge sig selv: Er den ultimative løsning i sådanne tilfælde ikke at lade real-time-processer køre på en dedikeret CPU, eventuelt under SMP?!

Fremtiden. Evolutions skedulering

Grundtanken er, at enhver proces er en udviklende proces med evnen til at tilpasse sig det miljø den lever i, hvorved at processens attributter bliver optimeret i forhold til bla. CPU-forbrug, I/O mv. Attributterne vil skifte værdi under afvikling, hvilket bliver styret af det, der betegnes som fintness

funktionen (mere om det senere). Disse optimerede attributter benytter evolutions-skeduleren til at finde den proces, der skal have næste kvantum af cpu-tid. Man kan sige, at man har indført en form for naturlig udvælgelse af processerne, således at det er den "stærkeste" der får mest cpu-tid.

Evolutions skeduleringen er delt op i to dele, køafvikling og procesafvikling, hvilket er afløsningen for traditionelle skeduleringsalgoritmer såsom Round Robin. I første fase benyttes evolutions-programmering til at vælge den optimale kø, som senere skal benyttes til at finde den "vindende" proces. Dernæst benyttes generiske algoritmer til at vælge den næste proces, der skal have CPU-tid. Som det illustreres i nedenstående figur, kan evolutionsskeduleren håndtere mere end to kø'er for hhv. Ready og blockedprocesser, hvis den er implementeret som 2D-skeduler.



Figur 6 2D Skeduler

Man definerer "*objektiv funktionen*" til at være et relativt styrkemål i forhold til problemområdet, som processen befinder sig i. Enhver proces er født med en objektiv funktion, der vil blive transformeret til processens fitnessfunktion og dermed processens fitnessværdi.
I evolutions skedulering defineres en "fitness" værdi til at være et heltal fra 0 til 255. Jo højere værdi des stærkere er processen i kampen med de øvrige processer. Fitness-funktionen der beregner værdien kan inkludere systemvariable såsom CPU-forbruget, I/O gennemstrømning, tiden, mv. Udover en fittnessværdi, skal der også være en "garanti"-værdi der skal garantere at processen vil få CPU-tid, og derved forhindre udsultning af processer. Denne garantiværdi vil blive mindre for hver gang processen får CPU-tid.

Umiddelbart ser det ud som om at der er mange fordele ved at benytte evolutions-skedulering, men der er dog en bagside af medaljen. Evolutions-skedulering tager tid! Dette er ikke fatalt meget og kan reduceres med mere simplificerede generiske algoritmer eller hurtigere cpu'er. Der forgår end del forskning på disse områder.

Kilder [6, 7]

Konklusion

Hovedkonklusion af vores behandling af dele af Linux-kernen følger her. Den er baseret på delkonklusionerne til de 4 hovedafsnit, hvorfor de vil blive ridset op her med tilføjelse af yderligere tanker, som de har givet anledning til.

Som nævnt under datastrukturer kan Linux-kernen kun afvikle 512 processer af gangen. Det har ikke været muligt at afgøre, om dette er en hæmmende faktor i den daglige afvikling af flerbruger-systemer – dertil er det nødvendigt at have indgående viden om de enkelte brugeres faktiske behov. Det står dog klart, at man aldrig kan imødekomme alle krav om brugervenlighed, og en form for afvejning af fordele og ulemper ligger derfor altid til grund for valg af datastrukturer. I Linux' tilfælde må nok hensynet til behandlingen af kørende processer have vejet tungt, idet disse behandles sekventielt, og et uendeligt antal processer således ville forøge behandlingstiden proportionalt. Man burde derfor nok opdele processerne i hensigtsmæssige strukturer, der nedsætter søgetiden af den næste proces, der skal afvikles.

Som nævnt under processkifte udskyder man færdigbehandlingen af visse systemkald. Dette er en effektiv måde at foretage andre og mere vigtige funktioner i et flerbrugersystem; at en brugerproces ønsker at læse data fra en fil betyder jo ikke, at han aldrig ombestemmer sig. Der er derfor ingen grund til at overføre data fra buffer cashe til brugeradresserum og derefter gøre denne til CURRENT. Men vigtigere er det faktum, at der meget muligt for indeværende er en højt prioriteret proces, der afventer CPU-tid, hvorfor man ikke vil forøge dennes ventetid med en behandling af andre processer, der måske slet ikke afvikles umiddelbart.
Yderligere skal det nævnes, at *wake_up ()* af processer, som er UNINTERRUPTIBLE i venten på en ressource, alle vil blive inkluderet i køen af kørende processer [3] og derfor selv skal afgøre om de kan fortsætte deres programafvikling. Det burde efter vores mening ikke være hensigten med en flerbruger-kerne, der i princippet er altvidende. Cornes er dog ikke helt enig med Rusling på dette punkt. Vi har desværre afspejlet tvivlen i vores behandling af processkift.

Som nævnt under prioritetsstyring er der muligvis en logisk fejl i behandlingen af real-time-processerne. Dette afspejler nok det faktum, at en real-time-proces sjældent vil blive afviklet under vilkår, hvor mange andre brugerprocesser influerer på dens afvikling. På trods af gængse flerbrugersystemers påviselige stabilitet, vil man nok på et hospitals hjerteafdeling foretrække at lade disse "real life" real-time-processer blive afviklet på en dedikeret CPU med adskillige back-up-systemer. Det har desværre ikke været muligt for os indgående at teste, hvorvidt prioritetsbehandlingen er fejlbehæftet. Skulle denne fejl imidlertid være reel, kunne det skyldes, at real-time-processers afvikling sammen med andre almindelige processer er en diciplin af akademisk natur og derfor har kunnet passere uden notits. Lad os håbe at testskørsler gør os til skamme i dette henseende.

Dog skal det understreges, at der tilsyneladende er umiddelbare fordele ved revidere proceskøer og lade real-time-processer ligge i deres egen kø. De kan behandles hurtigere, og skeduler behøver ikke lede efter nye real-time-processer ude i køen af kørende processer, ligesom de vil kunne blive behandlet før alle andre processer.
Men her har måske hensynet til de 3 forskellige prioriteringspolitikker spillet ind? Tilsyneladende kører de parallelt og under antagelse af, at det virker, skal de styre både real-time-processer og almindelige brugerprocesser samtidigt, hvilket kunne give anledning til unødigt system-overhead.

Som nævnt under evolutionsskedulering er der fordele ved at betragte processer som adaptive entiteter med behov og opførsel, der er skiftende og direkte afhængige af systemets nuværende kapacitet. Ligesom CISC-maskiner udviklede sig i retning af RISC, idet man begyndte at betragte applikationers faktiske forbrug af instruktioner, er det en naturlig udvikling af flerbrugersystemer' administration, at man betragter processers faktiske og historiske opførsel i det nuværende system for derved at gøre disse empiriske data til en parameter for tildeling af prioritet.
Det er klart, at der på denne måde er et forøget behov for administration – der er flere og større datastrukturer at behandle, og disse kan meget vel foranledige nye behov for behandling, der igen kan have afsmittende effekt på processers afvikling og prioritering. Men det er klart, at som der med tiden vil blive stadigt mere rå datakraft til rådighed, vil der også blive frigjort ressourcer til bedømmelse af den enkelte proces' afvikling således, at skeduler vil kunne forudbestemme pågældende proces' ressourceforbrug.

Litteraturliste

[1]. Running Linux, Matt Welsh & Lar Kaufman, O'reilly & Associates, inc 1997
[2]. Phil Cornes, Linux A-Z, Prentice Hall 1997
[3]. The Linux Kernel, David A Rusling , http://www.redhat.com/linux-info/ldp/LDP/tlk/tlk.htm
[4]. The Linux Way, Hanne Munkholm, Jesper Andreasen, Casper Troelsen, http://www.sslug.dk/artikler/kernel/
[5]. Evolution scheduling & Evolving processes, Jinlon Lin, http://www.iit.edu/~linjinl/esp.htm
[6]. Evolutionary Computations paper, Jinlon Lin mf. http://www.iit.edu/~linjinl/paper7.html
[7]. Open Source, Kenneth Geisshirt & Peter Toft, http://www.sslug.dk/artikler/OpenSource.htm
[8]. Source koden til Linux Red Hat 5.1 der er at finde på hvilken som helst distributions CD for Linux. Alternativt kan koden hentes på http://www.RedHat.com

Appendix

A Datastrukturer

Definitioner

Process tilstande

```
#define TASK_RUNNING          0
#define TASK_INTERRUPTIBLE    1
#define TASK_UNINTERRUPTIBLE  2
#define TASK_ZOMBIE           3
#define TASK_STOPPED          4
#define TASK_SWAPPING         5
```

Skedulerings politikker

```
#define SCHED_OTHER           0
#define SCHED_FIFO            1
#define SCHED_RR              2
```

task_struct

Each task_struct data structure describes a process or task in the system.

```
struct task_struct {
/* these are hardcoded - don't touch */
  volatile long        state;          /* -1 unrunnable, 0 runnable, >0 stopped */
  long                 counter;
  long                 priority;
  unsigned             long signal;
  unsigned             long blocked;   /* bitmap of masked signals */
  unsigned             long flags;     /* per process flags, defined below */
  int errno;
  long                 debugreg[8];    /* Hardware debugging registers */
  struct exec_domain   *exec_domain;
/* various fields */
  struct linux_binfmt  *binfmt;
  struct task_struct   *next_task, *prev_task;
  struct task_struct   *next_run,  *prev_run;
  unsigned long        saved_kernel_stack;
  unsigned long        kernel_stack_page;
  int                  exit_code, exit_signal;
  /* ??? */
  unsigned long        personality;
  int                  dumpable:1;
  int                  did_exec:1;
  int                  pid;
  int                  pgrp;
  int                  tty_old_pgrp;
  int                  session;
  /* boolean value for session group leader */
  int                  leader;
  int                  groups[NGROUPS];
  /*
   * pointers to (original) parent process, youngest child, younger sibling,
   * older sibling, respectively.  (p->father can be replaced with
   * p->p_pptr->pid)
   */
  struct task_struct   *p_opptr, *p_pptr, *p_cptr,
                       *p_ysptr, *p_osptr;
  struct wait_queue    *wait_chldexit;
  unsigned short       uid,euid,suid,fsuid;
  unsigned short       gid,egid,sgid,fsgid;
  unsigned long        timeout, policy, rt_priority;
  unsigned long        it_real_value, it_prof_value, it_virt_value;
  unsigned long        it_real_incr, it_prof_incr, it_virt_incr;
  struct timer_list    real_timer;
  long                 utime, stime, cutime, cstime, start_time;
/* mm fault and swap info: this can arguably be seen as either
   mm-specific or thread-specific */
  unsigned long        min_flt, maj_flt, nswap, cmin_flt, cmaj_flt, cnswap;
  int swappable:1;
  unsigned long        swap_address;
  unsigned long        old_maj_flt;    /* old value of maj_flt */
  unsigned long        dec_flt;        /* page fault count of the last time */
```

```
  unsigned long        swap_cnt;       /* number of pages to swap on next pass */
/* limits */
  struct rlimit        rlim[RLIM_NLIMITS];
  unsigned short       used_math;
  char                 comm[16];
/* file system info */
  int                  link_count;
  struct tty_struct    *tty;           /* NULL if no tty */
/* ipc stuff */
  struct sem_undo      *semundo;
  struct sem_queue     *semsleeping;
/* ldt for this task - used by Wine.  If NULL, default_ldt is used */
  struct desc_struct *ldt;
/* tss for this task */
  struct thread_struct tss;
/* filesystem information */
  struct fs_struct     *fs;
/* open file information */
  struct files_struct  *files;
/* memory management info */
  struct mm_struct     *mm;
/* signal handlers */
  struct signal_struct *sig;
#ifdef __SMP__
  int                  processor;
  int                  last_processor;
  int                  lock_depth;     /* Lock depth.
                                          We can context switch in and out
                                          of holding a syscall kernel lock... */
#endif
};
```

tq_struct

Each task queue (tq_struct) data structure holds information about work that has been queued. This is usually a task needed by a device driver but which does not have to be done immediately.

```
struct tq_struct {
    struct tq_struct *next;   /* linked list of active bh's */
    int sync;                 /* must be initialized to zero */
    void (*routine)(void *);  /* function to call */
    void *data;               /* argument to function */
}
```

B Skeduler Source Code:

```
asmlinkage void schedule(void)
{
        int c;
        struct task_struct * p;
        struct task_struct * prev, * next;
        unsigned long timeout = 0;
        int this_cpu=smp_processor_id();

/* check alarm, wake up any interruptible tasks that have got a signal */

        allow_interrupts();

        if (intr_count)
                goto scheduling_in_interrupt;

        if (bh_active & bh_mask) {
                intr_count = 1;
                do_bottom_half();
                intr_count = 0;
        }

        run_task_queue(&tq_scheduler);

        need_resched = 0;
        prev = current;
        cli();
        /* move an exhausted RR process to be last.. */
        if (!prev->counter && prev->policy == SCHED_RR) {
                prev->counter = prev->priority;
                move_last_runqueue(prev);
        }
        switch (prev->state) {
                case TASK_INTERRUPTIBLE:
                        if (prev->signal & ~prev->blocked)
                                goto makerunnable;
                        timeout = prev->timeout;
                        if (timeout && (timeout <= jiffies)) {
                                prev->timeout = 0;
                                timeout = 0;
                makerunnable:
                                prev->state = TASK_RUNNING;
                                break;
                        }
                default:
                        del_from_runqueue(prev);
                case TASK_RUNNING:
        }
        p = init_task.next_run;
        sti();

#ifdef __SMP__
        /*
         *      This is safe as we do not permit re-entry of schedule()
         */
        prev->processor = NO_PROC_ID;
#define idle_task (task[cpu_number_map[this_cpu]])
#else
#define idle_task (&init_task)
#endif

/*
 * Note! there may appear new tasks on the run-queue during this, as
 * interrupts are enabled. However, they will be put on front of the
 * list, so our list starting at "p" is essentially fixed.
 */
/* this is the scheduler proper: */
        c = -1000;
        next = idle_task;
        while (p != &init_task) {
                int weight = goodness(p, prev, this_cpu);
                if (weight > c)
                        c = weight, next = p;
                p = p->next_run;
        }

        /* if all runnable processes have "counter == 0", re-calculate counters */
        if (!c) {
                for_each_task(p)
                        p->counter = (p->counter >> 1) + p->priority;
        }
```

```
#ifdef __SMP__
        /*
         *      Allocate process to CPU
         */

         next->processor = this_cpu;
         next->last_processor = this_cpu;
#endif
#ifdef __SMP_PROF__
        /* mark processor running an idle thread */
        if (0==next->pid)
                set_bit(this_cpu,&smp_idle_map);
        else
                clear_bit(this_cpu,&smp_idle_map);
#endif
        if (prev != next) {
                struct timer_list timer;

                kstat.context_swtch++;
                if (timeout) {
                        init_timer(&timer);
                        timer.expires = timeout;
                        timer.data = (unsigned long) prev;
                        timer.function = process_timeout;
                        add_timer(&timer);
                }
                get_mmu_context(next);
                switch_to(prev,next);
                if (timeout)
                        del_timer(&timer);
        }
        return;

scheduling_in_interrupt:
        printk("Aiee: scheduling in interrupt %p\n",
                __builtin_return_address(0));
}
```

C Goodness Source Code

```
/*
 * This is the function that decides how desirable a process is..
 * You can weigh different processes against each other depending
 * on what CPU they've run on lately etc to try to handle cache
 * and TLB miss penalties.
 *
 * Return values:
 *       -1000: never select this
 *           0: out of time, recalculate counters (but it might still be
 *              selected)
 *         +ve: "goodness" value (the larger, the better)
 *       +1000: realtime process, select this.
 */
static inline int goodness(struct task_struct * p, struct task_struct * prev, int
this_cpu)
{
        int weight;

#ifdef __SMP__
        /* We are not permitted to run a task someone else is running */
        if (p->processor != NO_PROC_ID)
                return -1000;
#ifdef PAST_2_0
        /* This process is locked to a processor group */
        if (p->processor_mask && !(p->processor_mask & (1<<this_cpu))
                return -1000;
#endif
#endif

        /*
         * Realtime process, select the first one on the
         * runqueue (taking priorities within processes
         * into account).
         */
        if (p->policy != SCHED_OTHER)
                return 1000 + p->rt_priority;

        /*
         * Give the process a first-approximation goodness value
         * according to the number of clock-ticks it has left.
         *
         * Don't do any other calculations if the time slice is
         * over..
         */
        weight = p->counter;
        if (weight) {

#ifdef __SMP__
                /* Give a largish advantage to the same processor...   */
                /* (this is equivalent to penalizing other processors) */
                if (p->last_processor == this_cpu)
                        weight += PROC_CHANGE_PENALTY;
#endif

                /* .. and a slight advantage to the current process */
                if (p == prev)
                        weight += 1;
        }

        return weight;
}
```

D Fork Kildekoden

```
/*
 *  Ok, this is the main fork-routine. It copies the system process
 * information (task[nr]) and sets up the necessary registers. It
 * also copies the data segment in its entirety.
 */
int do_fork(unsigned long clone_flags, unsigned long usp, struct pt_regs *regs)
{
        int nr;
        int error = -ENOMEM;
        unsigned long new_stack;
        struct task_struct *p;

        p = (struct task_struct *) kmalloc(sizeof(*p), GFP_KERNEL);
        if (!p)
                goto bad_fork;
        new_stack = alloc_kernel_stack();
        if (!new_stack)
                goto bad_fork_free_p;
        error = -EAGAIN;
        nr = find_empty_process();
        if (nr < 0)
                goto bad_fork_free_stack;

        *p = *current;

        if (p->exec_domain && p->exec_domain->use_count)
                (*p->exec_domain->use_count)++;
        if (p->binfmt && p->binfmt->use_count)
                (*p->binfmt->use_count)++;

        p->did_exec = 0;
        p->swappable = 0;
        p->kernel_stack_page = new_stack;
        *(unsigned long *) p->kernel_stack_page = STACK_MAGIC;
        p->state = TASK_UNINTERRUPTIBLE;
        p->flags &= ~(PF_PTRACED|PF_TRACESYS|PF_SUPERPRIV);
        p->flags |= PF_FORKNOEXEC;
        p->pid = get_pid(clone_flags);
        p->next_run = NULL;
        p->prev_run = NULL;
        p->p_pptr = p->p_opptr = current;
        p->p_cptr = NULL;
        init_waitqueue(&p->wait_chldexit);
        p->signal = 0;
        p->it_real_value = p->it_virt_value = p->it_prof_value = 0;
        p->it_real_incr = p->it_virt_incr = p->it_prof_incr = 0;
        init_timer(&p->real_timer);
        p->real_timer.data = (unsigned long) p;
        p->leader = 0;          /* session leadership doesn't inherit */
        p->tty_old_pgrp = 0;
        p->utime = p->stime = 0;
        p->cutime = p->cstime = 0;
#ifdef __SMP__
        p->processor = NO_PROC_ID;
        p->lock_depth = 1;
#endif
        p->start_time = jiffies;
        task[nr] = p;
        SET_LINKS(p);
        nr_tasks++;

        error = -ENOMEM;
        /* copy all the process information */
        if (copy_files(clone_flags, p))
                goto bad_fork_cleanup;
        if (copy_fs(clone_flags, p))
                goto bad_fork_cleanup_files;
        if (copy_sighand(clone_flags, p))
                goto bad_fork_cleanup_fs;
        if (copy_mm(clone_flags, p))
                goto bad_fork_cleanup_sighand;
        copy_thread(nr, clone_flags, usp, p, regs);
        p->semundo = NULL;

        /* ok, now we should be set up.. */
        p->swappable = 1;
        p->exit_signal = clone_flags & CSIGNAL;
        p->counter = (current->counter >>= 1);
        wake_up_process(p);                     /* do this last, just in case */
        ++total_forks;
        return p->pid;

bad_fork_cleanup_sighand:
```

```
        exit_sighand(p);
bad_fork_cleanup_fs:
        exit_fs(p);
bad_fork_cleanup_files:
        exit_files(p);
bad_fork_cleanup:
        if (p->exec_domain && p->exec_domain->use_count)
                (*p->exec_domain->use_count)--;
        if (p->binfmt && p->binfmt->use_count)
                (*p->binfmt->use_count)--;
        task[nr] = NULL;
        REMOVE_LINKS(p);
        nr_tasks--;
bad_fork_free_stack:
        free_kernel_stack(new_stack);
bad_fork_free_p:
        kfree(p);
bad_fork:
        return error;
}
```