Innføring i operativsystemer:

Linux (eller Unix, som er det opprinnelige navnet på dette operativsystemet)

Det et operativt system gjør er nettopp gjør en maskin operativ, siker

Samling med programvare som ligger som et lag over hardware

Programvaren tilbyr grensesnitt mellom bruker og applikasjonene som nettleser osv…

Os har grensesnitt med hardware, effektiv – siker

Av eksperter for eksperter, programvareutviklere fleksibelt

C programmering brukes for operativsystemer, adresse minne og hardware

Ikke veldig teoretisk

2014 utgave av modern operating systems

Krav: java for eksempel, bruk av windows, tilgang til linux-system: linux-server, dual-boot windows/linux

10 av 12 godkjent arbeidskrav

Flervalg eksamen

Hva er et operativsystem?

Komponenter av moderne PC: cpu, minne, monitor, keyboard, usb, hard drive

Hovedoppgave: styring av maskinvare og (deling av) ressurser/prosessorer tilknyttet til en datamaskin er komplisert og vanskelig. Maskinvare, komponenter og programvare endres og forbedres raskt. Operativsystem (samling av programvare) håndterer dette, den holder maskinen oppdatert og sørger for at PC/nye hardware fungerer korrekt.

Trede oppgaver er abstraksjon vekk fra detaljer om maskinvare og systemprogramvare. Forholder oss til API’er (application programs) når vi programmer mot OS, metoder som kan kalles fra applikasjonens programkode.

Grensesnitt = Interface i form av kommandoer som utføres

OS Forenkler bruk, ferdig installert, lett oppdatering manuelt/ automatisk.

OS skjuler kompleksiteten av hardware. Designprinsipp: all maskinvare, komponenter og tilknyttede ressurser håndteres internt i OS.

Deloppgaver: håndtere prosesser og styre interprosesskommunikasjon (utveksle data) samtidig- programmer som kjører inkludert I/O, opprette/fjerne starte/stoppe prosesser, fordele mikroprosessorenes regnekraft mellom de aktive prosessene, prioritere prosesser på en effektiv måte, synkronisere noen prosesser,   
RAM- brukes for lagring av både data og programkode, er alltid begrenset ressurs, deles godt mellom alle prosessene når det kreves mer minne enn tilgjengelig(frigjøre plass), fungere både statisk ved oppstart av prosess og dynamisk mens prosessen kjører, beskytte det som er reservert for lesing prosess og mot skriving fra andre,   
filer, **nettverk**, komponenter og utstyr, sikkerhet og feilhåndtering.

Utstyrshåndtering: OS kommer med drivere- programkode for å håndtere ‘faste komponenter (CPU, RAM…)’ og ‘plug-ins (grafikk, **nettverk**, printere…)’, OS tilbyr SPI(Service Provider Interface): håndtere

Krav til godt konstruert OS:

Optimalisering med f.eks. effektiv – OS gjør ikke noe produktivt arbeid i seg selv, det meste av maskinens ressurser burde brukes til applikasjonene, OS har mange kompliserte og raske algoritmer for håndtering av datamaskinen som: scheduling- deling av noen få CPU’er på mange prosesser og paging and swapping- flytting av data og kode mellom RAM og disk

Vedlikeholdet- OS er stort programsystem eks: ‘guesstimate’ Widows 7 ca. 50 millioner kodelinjer, Linux-kjernen ca. 15 millioner linjer fordelt på 35k filer, stort antall utviklere involvert derfor må koordineringskostnader og -tid minimeres, OS må konstrueres modulært- god og logisk oppdeling i moduler, moduler bør vedlikeholdes uavhengig av hverandre

Korrekthet- feil i OS kan føre til at maskinen blir ubrukelig, derfor må det kunnes å håndtere tilfeldigheter og uforutsigbarheter og ha gode mekanismer for feilhåndtering, det krever -solid teoretisk fundament for implementasjon (utviklere som er erfaren), god modulisering internt i koden, godt og sikkert grensesnitt mot apps, gode og omfattende rutiner for testing og feilretting

Standardisering: følge standarder for samhandling med annen programvare og kommunikasjon med internett, velge ((s) av OS-designere) hvilke tjenester skal tilbys og hvordan de brukes, implementasjon av tjenestene i høynivå API som Java, hvordan filene organiseres, hvilken dataformater støttes (tegnsett og tall), hvordan skal eksekverbare programmer lagres

grensesnitt og abstraksjon: god for både brukere og utviklere, følge kjente og vanlige standarder, er stabilt og korrekt, skjuler den interne virkemåte av OS-et, ivaretar kompabilitet selv om maskinvare og systemprogrammer/drivere endres, kan sammenlignes med bruk av klasser i objektorientert programmering: klasseobjekter i Java tilbyr grensesnitt i form av public metoder som kan kalles fra ‘utsiden’, data som lagres i klassen og selve implementasjonen av metodene, inkapsles/skjules i den interne klasseobjektet, den defineres for brukerne med et grensesnitt som beskriver funksjonaliteten.

OS og systemtjenester: Systemtjenestene er tjenester som OS kan uføre på kommando ved å f.eks.: skrive data til disk, reserver en del av minnet… de utføres av OS når applikasjon gjør et systemkall ved å kalle en metode fra API’en, eksempel: når metoden **write** kalles vil den utføre en systemtjeneste som skriver data til en fil, OS sørger for at dette gjøres på riktig måtte uten at applikasjonen trenger å kjenne detaljene om hardware og drivere.

Kjerne i et OS: Alle systemtjenestene som tilbys til applikasjoner er samlet i det som kalles OS-ets kjerne (kernel), kjernen oppdateres oftest automatisk ved behov (ny maskinvare, forbedringer av OS-algoritmer etc.), kjernen er tilpasset maskinvaren, kjernen til OS er oftest skrevet i C (nærmest OS-et) eller direkte i maskinkode for effektivitet

Typer OS: mainframe(store maskiner/gamle dager), server(database, webserver), multiprocessor(tung/massive maskiner, raske beregner, vær, simulering), personal computer, handheld computer(mobil), embedded(klassesystemer i butikk, ABS-bremser i bil), sensor node(data fra vær), real-time(sanntid, ABS-bremser i bil), smart card(chipper).

Forskjellige krav til effektivitet.

IT og Linux-historie:

første generasjon 1945-55: Radiorør = vacuum tubes

imitation game, Alan Turing bombi, oversette tyske koder

ingen OS(menneskelig) manuelt med kode og data, ustabil, typisk brukt i militære

andre generasjon 1955-65: transistorer og bacth-systemer

mer stabile, kraftige maskiner, satsviskjøring- et program om gangen, kø-batch, fysisk hjelp, IBM FORTRAN-hullkort over til magnet

$FORTANT kompilere $LOAD laste in

Neste program vente helt til kjøring er ferdig, ved bruk av eksternt utstyr må CPU(prosessoren) vente, dårlig utnyttelse, et program i minne om gangen

integrerte kretsen(IC), raskere hardware og minne kom det multiprogramming and ICs i 1965-1980 som tredje generasjon, interaktive

kraftige maskiner, mer RAM= flere prosesser i minnet samtidig, når CPU venter kan den gi en annen prosess, ingen ‘idle time’,

multiprogramming utfører en context switch-bytter prosess, en prosses flere i minne ikke interaktive, kunne ikke snakke med maskinen mens den jobbet  
ikke flere aktive prosesser samtidig

ingen interaktivitet i batch og multiprogramming OS

IBM PC’er med fjerde generasjon i 1980-i dag

MULTICS ledene i å utvikle ekte interaktive OS, ikke vellykket men moderne OS inneholder: hierarkisk filsystem (mapper tre), grafisk grensesnitt med mus, tekstbehandlere, regneark og hypertekst(linker)

The mother of all demos

Interaktivitet oppnås med multitasking, Veldig mye context-switching- bytte prosessorer raskt virker som det skjer flere ting samtidig- timesharing: korte tidsintervaller

Internett 1968: ARPANET, forsvaret->åpen kildekode, Norge 1980

1969: Unix Ken Thompson jobbet i Multics, kildekode programmeringsspråk C, kraftig utvikling av Unix pga ARPANET: TCP/IP og GUI

Kommersielle UNIX, noen versjoner blir ikke kompatible, POSIX(portableOS) var en forsøk på løsning

Richard Stallman- GNU FSF: free software foundation, alt arbeid er frivillig og gratis, Videre utviklet emacs

utgangspunkt for LINUX(Linus Torvalds) utviklet for personal computers, la ut koden på en ftp-server, copylift lisens

I dag kan kjøpes i ulike distribusjoner (+ programvare), dominere på server og brukes av programutviklere – ikke kommet inn på desktopmarkedet

IBM PC’er kom med MS(MicroSoft), windows monopol: problemer: stabilitet, alvorlige sikkerhetsproblemer, oppleves omstendelig for utviklere, fordeler: industristandard, stort utvalg av programvære, lav nybegynner nivå

Femte generasjon i 1990-i dag med mobiler

User mode tillater ikke å gjøre fundamentale endringer på hardware.

OS i kernel mode har tilgang til kritiske funksjoner i hardware

Modusene gir beskyttelse mot lett forekommende krasj

Maskinvare/hardware: OS gjemmer detaljer vi ikke trenger å se

For å forstå OS må ha grunnleggende kjennskap til maskinvare- relevante grunnleggende prinsipper

Hovedkomponenter:

MMU- memory manegment unit håndterer minne og CPU (prosessor)

‘Hjernen’ som utfører det meste av beregningene for at maskinene skal fungere, inneholder: Mikroprosessor/prosessor kjerner- central processing unit/ processor cores -utfører beregninger parallellt

Er ofte koblet med en grafikkprosessor (GPU- graphical procesing unit)

Krever kjøling – frekvenser i nanos

Komponenter i en CPU: instruksjonssett, grunnleggende operasjoner/kode som kan utføres (ADD-addisjon, SUB-subtraksjon, MOV-move, JMP-jump, JEQ … etc.), hardkodet i elektronikken,   
for å få data for at operasjoner kan jobbe på de så er de i **registre**- dyre, raske minneceller som kan leses/hente data og skrives/ legge i data, få, brukes også til kontrolldata (f.eks. RAM-adressen til neste instruksjon)

Arithmetic Logic Unit (ALU, ‘execute unit’)- utfører alle regneoperasjoner og logiske tester, bruker registrene som operander og for å lagre resultater/bruker maskininstruksjonene for å utføre operasjonene

Pipeline/rør:

Minnekontrollenhet/ fetch unit- henter neste programinstruksjon fra minnet

Instruksjonsdekoder/decode unit- dekoder (setter opp) instruksjonene som skal kjøres i ALU

Ved flere prosessor kjerner lagres de i en holding buffer og distribueres videre – superscalar CPU, er mer effektiv

CPU-ens virkemåte: utfører et kompilert program (en og en maskin-instruksjon om gangen): oversatt fra et programmeringsspråk til en liste med maskin-instruksjoner som kan utføres, henter instruksjoner fra RAM, instruksjonen dekodes og legges i registrene, ALU utfører selve instruksjonen, kjører til alle instruksjonen er ferdig lest for programmet

Programutførelsen kan avbrytes midlertidig: avbrudd/interupts skjer ofte, utføres av OS, ved f.eks. behov for I/O eller feilsituasjoner, laster inn et annet program som håndterer avbruddet, en avbruddsrutine, etter avbrudd forsetter første programmet

Input/Output – usb/jack/hdmi, CPU kommuniserer bare gjennom å lese og skrive til/fra minne, derfor har celler med spesialisert minne og grensesnitt, brukes til kontroll og dataflyt fra/til disk, nettverk, tastatur, skjerm etc. Disken brukes som minne

Magnetiske disker/ moderne: solid state memory-bruker flash memory- raksere ikke enn RAM, ekstern enhet

Minne hastighet og kapasitet: registerne – 1nsec <1KB, cache-2nsec 4MB, RAM-10nsec 1-8GB, disk- 10msec 1-4TB

Minne kan også kalles: Hukommelse, arbeids-lager og hurtiglager på Norsk

RAM- random access memory – hovedminne, her lagrer vi programinstruksjonene som CPU skal kjøre og dataene som skal behandles av programmet mens den kjører, kan lese og skrive data til -aksesseres ved å indeksere, ikke permanent- data forsvinner når maskinen slåss av

Kretskort med minnebanker/memory-banks, mange minneceller ligger i brikkene – de har adresserbare lagerceller (8-bit tradisjonelt, minste enheten), hver celle har en heltallsadresse som kan brukes til å lese verdien eller lagre/skrive en verdi- det skjer ved å overføre data på databussen, (kan skrives fram og tilbake) mellom minnecellen og det som ligger lageret eller skal lagres i et register i CPU, det er en egen enhet i CPU, en MMU som håndtere dette

Overføring av data mellom RAM og CPU over bussene tar lang tid, derfor har moderne CPU’er et cache-minne (depot,forråd) som er mindre samlinger av raske minneceller, liger internt i/nærmere enn RAM CPU, bruker ikke buss-systemet isteden hentes direkte fra cache, styring håndteres i egen cache-kontroller internt i CPU, som er transparent for programkoden som utføres (koden ser ikke om det hentes fra cache eller minne). Cache-algoritmer er kompliserte og de krever my ordning og testing for å sikre korrekthet og konsistens

Hovedproblemer: må ha cache-algoritmer for å løse: når du skal lagre nytt element i cache, på hvilken linje, hvilket element skal du ta ut når du trenger plassen, hvor i hovedminne legger du det elementet du tok ut

Ulike typer/modeller cache- for å få effektiv bruk: nivå 1/L1- eget lite minneområdet internt i en CPU-kjerne, nivå 2/L2- større ekstern eller intern cache-brikke med egen rask buss, en egen cache for hver CPU-kjerne, nivå 3/L3 - ‘mater’ nivå 2 med data fra RAM, deles alle kjerene til og med noe av GPU (kan også bil raskere hvis den har en egen cache-system)

ROM- read only memory – permanent minne, innholdet ligger fast i elektronikken, lagrer BIOS (Basic Input Output System)- inneholder instruksjoner som kjøres når maskinen startes (booting/bootstrapping- løfte seg selv), er nødvendige for å kunne få systemet til å starte, noen linjer med kode som kjøres og leser OS fra disk og laster opp OS i RAM, varianter: Firmware og UEFI

Busser- kommunikasjonsmedium som flytter data fra en hardware komponent til en annen, kan være internt eller eksternt, parallelle (signalveier mellom komponenter) buss-systemer kommuniserer mellom sentrale interne deler av datamaskinene, som forbinder CPU med RAM og I/O-celler (ekstern kommunikasjon)

Arkitektur med tre ulike busser med sine egne oppgaver:

databuss- overfører data (programinstruksjon/data for programmet) mellom CPU og minne, hvis CPU skal ha data som skal inn i registre, overføres binært som elektriske signaler (parallelle ledere- minst 8 evt. 16, 32, 64 som sender 0 og 1, en bit på hver leder), CPU, RAM og I/O-celler kan lese og skrive til databussen, kun en enhet om gangen kan legge ut en dataverdi på bussen – viktig for å være konsistent og ikke få feil, CPU styrer kommunikasjonen på databussen og adressering av minne ved å sende signaler på adressebussen

adressebuss- bare CPU kan legge en verdi på adressebussen, alle andre kun leser, utenom I/O til disk som bruker adressebussen, før overføring av verdi på databussen legger CPU ut minneadressen som skal brukes på adressebussen, egne dekodingskretser leser denne adressen og sørger for at riktig minnebrikke kobles inn/chip select

kontrollbuss: for å angi om en verdi skal leses eller skrives fra/til minne og for å synkronisere, styrer samarbeidet mellom CPU og maskinvare f.eks: angir om verdi på databussen skal leses fra eller skrives til minneadressen på adressebussen/write-enable, synkronisering av CPU(er) på samme maskin og minneceller slik at vi sikrer korrekt overføring av verdier på bussene, overfører signaler fra I/O-enheter om at de har eller vil ha data eller om at interrupt har skjedd, synkronisering av I/O til/fra disk. I/O-enheter og komponenter kan sende avbrudd-signaler over kontrollbussen til CPU for å varsle om en hendelse som må håndteres ved å starte et program som håndterer, program som blir avbrudd kan være av f.eks.: -tastetrykk, museklikk/bevegelse, innkommende data over en nettverksport\I/O-celler, hardwarefeil. Det skjer også planlagte avbrudd av timere flere ganger i sekundet for at timesharing skal være mulig i\av OS. Avbruddsignalene har en ID-nummer (i avbruddsvektorer/arrays) som har årsaken til avbruddet i seg, når CPU mottar det: fullfører den de nåværende instruksjon, deretter startes det en avbruddsrutine (program som håndterer avbruddet), etter gjenopptar forrige programmet hvor den vær stoppet. Systemstatus må lagres før avbrudd håndteres.

CPU har tilgang til alle tre busser, leser og skriver fra kontroll og data bussen, bare skriver til adressebussen

RAM- skrive lese databussen og kontrollbussen, bare lese fra adressebussen

I/O- kan skrive og lese fra alle

En form for avbrudd skjer når I/O vil lese fra eller skrive til RAM. Når du driver med Input/Output så er ikke CPU nødvending, derfor har PC’er en DMA-brikker (Direct Memory Access) for å gjøre overflytting av data raskere, de kan legge på adresser på adressebussen og slå av/på write enable i kontrollbussen, konflikten med DMA og CPU er at DMA kan overskrive data på adressebussen pga. begge har tilgang til å skrive/lese data for å unngå har vi: Burst-mode der DMA stopper CPU med et avbruddsignal(IRQ), eller Cycle stealing: DMA bruker bussen når CPU ikke gjør det.

En hel buss-syklus for å kjøre en instruksjon i CPU består av (forenklet) operasjoner: CPU legger en adresse på adressebussen, CPU legger styringsdata på kontrollbussen, verdi overføres mellom to enheter på databussen  
I raske moderne systemer kan det typisk ta et hundre-milliondels sekund, vi har da en 100-MHz systembuss -100 millioner sykluser i sekundet

Buss-systemet blir en flaskehals(bottelneck) i en rask datamaskin: teoretisk ytelse for typisk systembuss: 100-MHz med 32-bits busser, 4 bytes med data kan sendes til/fra CPU 100 millioner ganger i sekundet, CPU jobber mye raskere enn dette: prosessorkjernene er typisk i GHz-området, 1GHz = 109 = 1000 millioner instruksjoner per sekund/avløsinger av registre-1nsec/klokketikk, kan forbedres ved bruk av cacheminne

Prosess: en program som kjøres på datamaskin, er en modell/abstraksjon av det fysiske programmet som eksekveres i hardware, er sekvensiell(kjører til den er ferdig): formål:-løse oppgave -prosessor/CPU(er) -minne, disk, kontrolldata -programkode  
program er statisk enhet som ligger lagret som kildekode/maskinkode, koden forandrer seg ikke når programmet kjører prosess er dynamisk enhet som endrer seg hele tiden under kjøring av programmet, registre og andre enheter i CPU oppdateres mange millioner ganger i sekundet, variablene som er lagret i RAM endrer verdier, filer åpnes skrives til og lukkes  
begrepene program, prosess og tråd brukes om hverandre og upresist  
OS definere prosess som en samling av ressurser som er nødvendige for å utføre en oppgave beskrevet med programkode på en datamaskin, ressursene som utgjør en prosess: eget minneområde som er reservert for prosessen, inneholde i CPUens registre, programkoden: ferdig kompilerte maskininstruksjoner som er lastet inn i RAM, programteller som hele tiden holder rede på funksjonskall når programmet kjøres, filer og I/O-enheter som skal leses/skrives og andre kontrolldata som OS bruker for å håndtere prosessene

Andre deler av OS’et avhenger av prosessene, Prosessene gjør det mulig for datamaskinen å "gjøre flere ting samtidig" -- "(pseudo) concurrency" – selv om den bare har én CPU!  
Eksempler på "pseudoparallelism": – Web-servere og I/O – GUI, virusscanner og andre "bakgrunnsprosesser"

●"Multiprogramming" – flere prosesser i minnet samtidig  
● Kun én prosess kan kjøres i CPU'en om gangen  
● Prosessene som ikke kjører ligger i OS'ets eksekveringskø  
● CPU-en bytter på å utføre prosessene i køen, typisk 50-100  
ganger i sekundet (timesharing)  
● En prosess kjøres i et kort tidsintervall, typisk noen ms  
● Ved utløp av hvert intervall tar OS'et ut kjørende prosess og  
velger neste prosess fra køen som skal kjøres (scheduling)  
● Hvis tidsintervallene er korte og OS/CPU effektive nok,  
oppleves det som om maskinen gjør flere ting samtidig

Four principal events that cause processes to be created: 1. System initialization. 2. Execution of a process creation system call by a running process. 3. A user request to create a new process. 4. Initiation of a batch job.

Alle prosesser lages ved at en allerede eksisterende (og kjørende) prosess oppretter en ny prosess\*● Prosessen opprettes ved gjøre et systemkall som starter en systemtjeneste i OS-kjernen ● Prosesser kan startes i forgrunnen: – En interaktiv prosess som kommuniserer med bruker ● Eller i bakgrunnen: – En prosess som ikke trenger input fra bruker – F.eks. en web- eller mail-"daemon" i Linux, som "våkner" opp når det kommer en "request" som må håndteres  
\*: Med unntak av den aller første prosessen (init i Linux)

En prosess i Linux som oppretter en ny prosess blir forelder til den nye prosessen  
● Prosessen som opprettes er barn av forelder-prosessen  
● OS lagrer alle barn-forelder forhold mellom prosessene  
● En forelderprosess kan vanligvis styre, stoppe, starte og  
slette sine egne barneprosesser  
● Alle barn har bare én forelder, men en prosess kan ha  
mange barn  
● Gir en ordning av prosessene i et hierarkisk tre

Spawn\*: Opprettelse av en Linux-prosess  
● I Linux brukes begrepet “spawning” om det som skjer når en forelderprosess oppretter en barneprosess  
● Spawning gjøres av shellet eller av en annen applikasjon, (vanligvis) ved å utføre C-systemkallet fork() i Linux-kjernen  
● fork() og de andre systemkallene for spawning er dokumentert i man-systemet (Section 2: System Calls)  
\*: spawn: to produce;

Linux-systemkallene fork og vfork  
● Oppretter en barneprosess som er en kopi av forelder-prosessen, inkludert programkoden,  
innholdet av RAM, registre, filer, og prosessdata  
● fork: – Barneprosessen får et eget minneområde i RAM  
● vfork: – Barne- og forelderprosess deler minneområdet. Forelder er blokkert inntil barneprosess avsluttes

execvp – for å starte nye programmer  
● Systemkallet execvp erstatter nåværende prosess med en ny prosess\*  
● Hele innholdet av memory for prossesen som kaller execvp slettes, og byttes ut med et nytt program som leses inn fra disk  
● Vanlig å først gjøre en fork, og deretter la den nye barneprosessen gjøre execvp for å bytte ut seg selv med et nytt program som skal startes

\*: execvp er et av flere beslektede systemkall i Linux som kan starte nye prosesser

init (systemd): Den aller første prosessen som startes i Linux etter  
booting, I nyere Linux-systemer heter oppstartprosessen gjerne systemd – "system daemon" – i stedet for init, Opprettes av Linux-kjernen, som igjen startes av ROM-kode, init/systemd er “foreldreløs”, alle andre prosesser har en forelder

prosess-kjeder i Linux: init er stamfaren til alle andre prosesser på systemet, Kan alltid dannes en kjede av prosesser fra nåværende prosess til forelder, til forelders forelder osv. helt opp til init

Typical conditions which terminate a process: 1. Normal exit (voluntary). 2. Error exit (voluntary). 3. Fatal error (involuntary). 4. Killed by another process (involuntary).

Prosessenes tilstand: I en prosess' "levetid" – fra den opprettes til den  
termineres – har den hele tiden en tilstand (process state), Tilstanden vil endres etter hvert som eksekveringen og tiden går fremover, Prosessen vil ofte ikke være "kjørende inne i CPU'en", men i er i stedet i en "ventetilstand"  
To hovedårsaker til venting:  
– Mange prosesser som skal dele på én CPU  
– "Noe skal skje" før prosessen kan gå videre – oftest  
venting på input fra en annen prosess eller I/O-enhet

Three states a process may be in: 1. Running (actually using the CPU at that instant). 2. Ready (runnable; temporarily stopped to let another process run). 3. Blocked (unable to run until some external event happens). 1. Process blocks for input 2. Scheduler picks another process 3. Scheduler picks this process 4. Input becomes available

Forenklet systemmodell: "Alt er prosesser": Brukerprogrammer, systemprogrammer, I/O, drivere... Alt som utføres i maskinen er prosesser som er "running", "ready" eller "blocked", Interrupts betrakts som hendelser som blokkerer én prosess (f.eks. den som venter på I/O) og starter en annen (avbruddshåndtering for f.eks. å lese/skrive I/O), I/O blir da prosesser som blokkeres inntil I/O-interrupt skjer, og blokkeres igjen når I/O er ferdig, Scheduleren blir et "fundamentalt" nivå som har ansvaret for å starte, stoppe og blokkere prosesser, både ved faste interrupts (timesharing) og tilfeldige (I/O m.m.)

Implementasjon av prosessmodellen: Alle dataene som OS'et trenger å kjenne til for å håndtere én bestemt prosess, lagres i en fast datastruktur som kalles en prosessdeskriptor eller "process control block", fast datastrukur som inneholder alt som OS trenger å vite om prosess for å håndtere prosessene korrekt og det er en dynamisk datastruktur, Inneholdet i denne datastrukturen endres hele tiden etter som programmet kjøres og tiden går.(snapshot)  
Deskriptorene for alle prosessene som finnes på maskinen lagres i prosesstabellen i OS-et  
Prosesstabellen er en array der hvert element er en deskriptor (en struct i C) for en bestemt process, Prosesser i Linux har en unik Process ID (PID) som er indeksen til prosessens deskriptor i prosesstabellen

Håndtering av interrupts/scheduling: Bytte av kjørende prosess i CPU skjer typisk hundrevis av ganger i sekundet pga. avbrudd/timesharing, Hele datastrukturen/deskriptoren til en prosess må lagres unna i prosesstabellen ved avbrudd, når status går fra running til ready eller blocked. Prosessen må etterpå kunne hentes frem igjen og gjenskapes, slik at den fortsetter å kjøre med nøyaktig samme verdier for prosessdataene som da den ble lagret, Lagring av data for kjørende prosess og innhenting av data for neste prosess som skal kjøre, gjøres av "superraske" rutiner som er "hardkodet" i elektronikken eller programmert i lavnivå-språk (assemblerkode).

Effektivitet av flerprosessmodellen: Vil finne ut hvor godt systemet klarer å utnytte CPU'en. Når prosesser venter på I/O (er "blocked"), vil systemet fortsette å kjøre prosessene som er "ready". Hvis alle prosessene som kjører på systemet venter på I/O samtidig, vil CPU'en ikke ha noe å gjøre (være "idle"). Vi kan måle effektiviteten ved å prøve å finne ut hvor stor  
del av tiden som CPU ikke er "idle": – Kalles "CPU utilization" (utnyttelsesgrad) – Måles i prosent av den totale tiden på systemet – F.eks. vil en "CPU utilization" på 85% bety at prosessoren er "idle" i 15% av tiden

Vi antar at utnyttelsen av CPU avhenger kun av to ting: – Hvor mange prosesser som ligger i RAM samtidig – Hvor stor prosent av totaltiden for en prosess som brukes på å vente på I/O (er i "blocked" tilstand)  
For prosessmodellen gjør vi noen (grove) forenklinger: – Alle prosessene bruker samme andel p av tiden til å vente på I/O (0 ≤ p < 1) – F.eks. betyr p = 0.6 at prosessene er blokkert for I/O 60% av tiden – Venting på I/O skjer på helt tilfeldige tidspunkter – Alle prosessene utføres helt uavhengig av hverandre

La n være antall prosesser i RAM og p andelen med tid som hver prosess bruker på å vente på I/O, Med antagelsene på forrige lysark, vil sannsynligheten for at alle de n prosessene venter på I/O samtidig på et gitt tidspunkt (slik at CPU er "idle") være pn  
Med f.eks. n = 5 prosesser som venter på I/O 60% av tiden, vil CPU i gjennomsnitt være "idle" i 0.65 ≈ 0.08 = 8% av tiden. Utnyttelsesgraden for CPU er da 1 – 0.08 = 0.92 = 92%  
Formel: CPU utilization = 1 – pn

– Effektiviteten bli bedre når det er flere prosesser i minnet samtidig  
– Mere RAM (som har plass til flere prosesser) gir derfor bedre utnyttelse av CPU  
– Forbedringen i effektivitet som mere RAM gir "flater ut" ganske raskt, fordi CPU etterhvert blir nesten 100% effektiv (er nesten aldri "idle")  
– For å forbedre effektiviteten ytterligere må det da settes inn flere CPU'er som kan jobbe parallelt

Kjøring av bakgrunnsprosesser: Prosessen kjøres samtidig med shellet, Shellet venter ikke til kommandoen er ferdig, men er klar for å motta en ny kommando med en gang, Shellet vil skrive ut et jobbnummer og PID for bakgrunnsprosessen som opprettes, Bakgrunnsprosesser kjøres med lavere prioritet, Kan ikke kommunisere interaktivt med shellet eller med bruker

Tråder(“thread/path of execution/control") er en av ressursene som utgjør en prosess:  
– Utførende, sekvensiell enhet som går gjennom programkoden og eksekverer maskininstruksjoner. En prosess kan inneholde én eller flere tråder som kan kjøre parallelt (”samtidig”) i ulike(!) deler av prosessens programkode. Tråder kan kjøre uavhengig av hverandre, men mange problemer krever at trådene synkroniseres.

Prosess: Formål: Løse oppgave  
● Tråder  
● Minne, disk, kontrolldata  
● Programkode

Tråd vs. prosess  
Trådene er “lettvekts-prosesser”: – Raskere å opprette en ny tråd enn en ny prosess – Raskere kommunikasjon mellom samarbeidende tråder inne i en prosess, enn mellom ulike prosesser  
● Parallelle tråder er mer utfordrende for programmereren: – Trådene deler minnet og andre ressurser som er allokert til prosessen, mange mulige kilder til feil og inkonsistens  
– Samarbeidende tråder må synkroniseres i koden – Krever ofte "fintenking" og nøyaktighet i design for å få multithread-kode til fungere korrekt og effektivt

Single- og multi-thread OS: ● Tidlige OS: Single-thread: – Prosesser med bare én “tråd” – Begrepet “threads” fantes ikke, bare separate prosesser  
● Moderne OS: Multi-thread: – Flere tråder i hver prosess – Ble vanlige for ~25 år siden   
– Spesielt fremveksten av internett skapte behov for multi-thread OS – Både Windows og Linux er multi-thread

Hvorfor bruke flere tråder? Prosesser med bare én tråd stopper når de må vente på en  
ressurs, f.eks. I/O eller en annen maskin på nettet. Tråder gir mulighet til å la programmet fortsette å gjøre nyttig arbeid i deler av koden som ikke er avhengig av ressursen som det ventes på. Tråder gir bedre effektivitet: – Raskere å opprette enn hele prosesser – Gir bedre CPU-utnyttelse og mindre "idle-tid" / venting på I/O – Utnytter parallelle maskiner med flere kjerner/CPU'er bedre

Eksempel: Tekstbehandlere  
● Mye I/O hele tiden: – Dialog med bruker: Kommandoer og museklikk – Stadig lagring av backup/siste endringer  
● Beregningstungt: – Dokumentet skal hele tiden reformateres når det endres – Mange sider og mye data som stadig skal endres  
● Kan implementeres effektivt med tre synkroniserte tråder som deler samme data og filer:   
1. Brukerdialog 2. Disk I/O 3. Formatering/beregninger

Eksempel: Web-servere: Brukerklienter sender "requests" over en nettverksforbindelse til  
en serverprosess, for å hente websider (og annet innhold). Websider kan ligge i RAM (web-cache) eller på disk. Med bare én tråd må prosessen vente på at I/O skal bli ferdig hver gang en side ligger utenfor cache – serveren blir treg. Løsning: Egne tråder som håndterer henting av websider

Eksempel: Store datamengder  
Prosessen skal lese inn store datamengder fra disk, gjøre beregninger og skrive ut igjen prosesserte data til disk. Kan bruke bare én tråd, som leser inn en og en blokk med data,  
utfører beregninger og skriver ut resultater, før neste blokk leses. Problem: CPU "idle" ved I/O, mens mange beregninger gjenstår  
Løsning med (minst) tre tråder og to buffere: 1) Input-tråd leser data fra disk inn i et inn-buffer 2) Beregningstråd leser inn-buffer, beregner, skriver til et ut-buffer 3) Output-tråd leser fra ut-buffer og skriver til disk  
Fungerer hvis I/O bare blokkerer tråden og ikke hele prosessen

Prosess- og tråd-data: Hvis det er flere tråder i en prosess, vil de dele samme minneområde, programkode og I/O-enheter  
Tråder må også ha egne kontrolldata som holder rede på: – Hvor i programkoden tråden befinner seg – Hvilken maskininstruksjon som er den neste som utføres – Hvilke verdier i minnet som skal brukes av neste instruksjon – Kjeden/stacken av funksjonskall som til enhver tid er aktive  
Trådens kontrolldata lagres både i registervariable og i en egen eksekveringsstack\* når den kjøres  
\*: Hver tråds stack må ha et eget minnområde, som ofte vil ligge i CPU-cache fordi det brukes mye

Lagring av prosesser og tråder i OS: Dataene for hver prosess lagres i en prosess-deskriptor. Deskriptorene for alle prosessene som finnes på maskinen lagres i prosesstabellen i OS-et. Dataene for trådene i en prosess lagres i en tråd-deskriptor  
Tråd-deskriptorene kan lagres lokalt i prosessene→ Lagring av trådinfo. kan også gjøres på andre måter, avhengig av hvorledes (og hvor i systemet) tråd-håndteringen implementeres

Thread States: 1. Thread blocks for input 2. CPU switches to another thread 3. This thread starts executing in the CPU 4. Input becomes available   
Both threads and processes can be in running, blocked, or ready state.

Hvor ligger håndteringen av tråder i systemet? -Tråder kan implementeres i kjernen, internt i OS'et. Eller utenfor OS'et, der all annen software kjøres. Eller som en hybrid implementasjon, der begge deler av systemet er involvert.

Kjernetråder og brukertråder  
● Kjernetråder/“Kernel-level threads” /”Native threads”: – Tråder som opprettes og håndteres av OS-et selv – All trådhåndtering gjøres inne i OS-kjernen med systemkall  
Operativsystemets eget API tilbyr systemkall (til f.eks. C-programmere) for å opprette nye tråder inne i en prosess. Alle trådene er kjente for OS'et, som selv håndterer scheduling, interrupts og timesharing for både tråder og prosesser. OS håndterer også sending av meldinger og data mellom tråder og prosesser gjennom systemkall i API'et. All informasjon (kontekst) for en prosess og alle de individuelle trådene inne i prosessen er lagret i OS-kjernen  
Fordeler ved kjernetråder: OS-et kan utnytte maskiner med flere prosessorer/CPU-kjerner effektivt ved å kjøre kjernetråder (ekte) parallellt. Viktige OS-rutiner i kjernen kan også gjøres multi-thread og parallelle(!). OS kan enkelt prioritere f.eks. tunge prosesser med mange tråder, gir bedre utnyttelse av prosessorene. Viktig: Kjernetråder er blokkeringsfrie, dvs. at en tråd som f.eks. venter på I/O ikke stopper andre tråder i samme prosess fra å kjøre \*: Dette helt nødvendig for at f.eks. webservere og tekstbehandlere kan bruke tråder effektivt.  
Ulemper ved kjernetråder: Håndtering av kjernetråder er generelt mer tidkrevende enn  
å håndtere brukertråder. Opprettelse av en kjernetråd krever avbrudd i CPU og oppstart av systemrutiner for å håndtere prosesstabell m.m. Hver gang en tråd skal starte å kjøre, må det gjøres en full "context switch" (dvs. bytte av kjørende tråd) i kjernen. "Context switch" er ressurs- og tidkrevende, bl.a. fordi CPU og cacheminne må "flushes" hver gang

● Brukertråder/“User-level threads”/“Green threads”: – Tråder lages og håndteres av et runtime-system som f.eks. Java Virtual Machine (JVM) – All trådhåndtering skjer utenfor OS-kjernen.   
Trådene håndteres av et programvarebibliotek som kjører i user space, utenfor OS-kjernen: OS-kjernen håndterer bare vanlige single-thread prosesser, og "vet ingenting om" trådene. All trådinformasjon lagres internt i trådbiblioteket. Trådbiblioteket har eget run-time system som håndterer "alt": – Opprettelse og fjerning av tråder – Sending av data og meldinger mellom trådene – Schedulering av tråder.   
Fordeler ved brukertråder: Programmene kan kjøres på "alle" OS. Ofte enklere API, høyere abstraksjonsnivå enn kjernetråder. Brukertråder er raskere å opprette, håndtere og terminere. Bytte av kjørende tråd i runtime er mye raskere enn avbrudd og fullt "context switch" i kjernen. Hver prosess kan kjøre "skreddersydd" scheduling. Skalerer bedre, mer plass og raskere håndtering av data i user space enn i kjernen når antallet tråder blir svært høyt.   
Ulemper ved brukertråder: Viktig: Brukertråder er ikke blokkeringsfrie. OS-kjernen "ser" bare én single-thread prosess og ikke de ulike brukertrådene inne i prosessen. En tråd som gjør et blokkerende systemkall (typisk I/O eller en "page fault") vil derfor stoppe alle trådene i prosessen. Kan "fikses" med å bruke ikke-blokkerende I/O eller ved å skrive "wrapper-kode" rundt I/O som sjekker om en tråd kan/bør blokkere eller ikke – men det skaper en masse andre problemer som vi ikke vil ha. Brukertråder kan ikke utnytte flere prosessorer/kjerner, fordi OS ikke "vet om" at trådene finnes. Run-time systemer for håndtering av brukertråder utfører  
ikke timesharing med jevne "klokkeavbrudd". Brukertråder vil derfor ikke automatisk "gi fra seg CPU" til andre tråder i samme prosess. Bibliotek for brukertråder må ha funksjoner som gir trådene mulighet til å gi opp CPU "frivillig", slik at alle tråder kan få kjøre selv om det ikke gjøres timesharing på brukernivå. \*: OS-kjernen gjør typisk dette flere hundre ganger per sekund for å dele CPUen mellom prosessene.

Hybride trådimplementasjoner: Forsøker å utnytte fordelene ved både bruker- og kjernetråder ved å kombinere dem i implementasjonen. En mulig løsning er å multiplexe flere brukertråder ned på én kjernetråd: – Brukertrådene i en prosess kjøres av et mindre antall  
kjernetråder – "mange-til-én mapping" – Grupper på f.eks. tre brukertråder kjøres av hver sin kjernetråd – Blir fortsatt flere raske switcher mellom brukertråder enn de langsommere mellom kjernetråder – Samtidig unngås det at blokkerende systemkall stopper alle trådene i hele prosessen.

En annen løsning: Scheduler activations: Prøver å utnytte kjernetrådenes gode egenskaper, men beholde hastigheten i brukertråder. Spesielt prøver man å unngå at brukertråder blokkerer en hel prosess når det gjøres I/O. Gjøres ved at kjernen tilbyr brukertrådenes runtime-system  
virtuelle prosessorer som de kan kjøre på. For å fungere, kreves det at kjernen får gjøre et upcall for å fortelle runtime-systemet når en brukertråd ikke lenger er blokkert og kan kjøre igjen – bryter mot den tradisjonelle lagdelingen i et OS. Relativt komplisert, men effektivt.

Pop-up threads i servere: Rask håndtering av requests/meldinger. Creation of a new thread when a message arrives.

IPC: pipes

Hvorfor trenger vi IPC? Det vil alltid være prosesser (og tråder) som trenger å sende data til/fra andre prosesser/tråder\*  
Med IPC kan vi unngå at prosesser ødelegger for hverandre: – Eksempel: Reservasjon av flyseter  
Prosesser kan avhenge av hverandre og derfor måtte kjøres i riktig rekkefølge – IPC kan brukes til synkronisering: – Eksempel: Prosess B skal skrive ut resultatene som beregnes  
av prosess A  
\*: Datautveksling er enklere mellom tråder enn mellom prosesser, fordi trådene deler samme minneområd

Race condtion/hazard: Feil eller ukontrollert utfall avhengig av hvilken rekkefølge prosessene kjører.

Race conditions og timesharing/scheduling: Feil pga. race conditions oppstår ofte fordi prosessene/trådene avbrytes med jevne mellomrom (klokkeavbrudd) for at de skal dele på CPUen(e): – Prosess 1 leser eller skriver til en delt minne, og skal deretter utføre en beregning som avhenger av at det delte minnet inneholder korrekte verdier – Prosess 1 avbrytes av OS for at prosess 2 skal få kjøre, før prosess 1 har startet beregningen – Prosess 2 endrer på verdiene i det delte minnet – Når prosess 1 starter igjen, vil beregningen kunne gi feil resultater fordi kritiske verdier i delt minne er endret

Eksempel: Et spooling system for printere: En printer er en delt ressurs for brukere og prosesser. Spooling brukes for at prosesser ikke skal måtte vente på å få tilgang til printer:   
-Alle prosesser som skal printe/skrive ut en fil, lagrer bare navnet på filen i en delt systemressurs, en "spool-katalog" – Spool-katalogen inneholder en liste/kø med filer som venter på utskrift – Selve printingen håndteres av en bakgrunnsprosess i systemet, en printer daemon, som "sover" når printerkøen er tom – Printer daemon leser neste printjobb fra spool-katalogen, utfører denne og sletter jobben fra spooling-systemet

Hvordan unngå race conditions? Vi må unngå at to prosesser kan lese og skrive delte ressurser (minneområde/data, filer) "samtidig". Kan gjøres ved at OSet tilbyr en mekanisme for gjensidig utelukkelse ("mutual exclusion") til programmerere: – Når én prosess bruker en delt ressurs, kan alle andre prosesser utelukkes fra tilgang til denne ressursen – Vil f.eks. løse spooling-problemet på forrige side: Prosess B får ikke endre den delte variabelen in før prosess A er ferdig med å bruke den. Valg av design og implementasjon av mutual exclusion er et viktig element i utviklingen av et OS

Scheduler og scheduling-algoritmer: Multiprogramsystemer har flere prosesser i minnet samtidig, Alle prosesser som er i "ready"-tilstand "konkurrerer" om å få bli den neste som skal få kjøre i CPUen?, Det må gjøres et valg: Hvilken prosess skal få kjøre? \* ,Scheduleren er den delen av OS som gjør dette valget, ved å bruke en scheduling-algoritme  
\*: I multi-thread systemer vil scheduleren oftest håndtere valg av tråder i stedet for hele prosesser – hvilken tråd i ready-tilstand er den neste som får kjøre?

Når gjør et OS prosess-scheduling? Når kjørende prosess selv oppretter en ny prosess: Kjøre  
barneprosessen først eller fortsette å kjøre forelder i stedet? Når kjørende prosess terminerer: Velg en annen prosess fra ready-køen, Når kjørende prosess selv blokkerer, f.eks. for I/O eller ved å gjøre down/wait på en semafor eller betinget variabel, Når kjørende prosess stoppes av det regelmessige klokke-avbruddet i timesharingen, Andre avbrudd fra hardware, spesielt I/O-enheter, kan også føre til scheduling

Prosess-scheduling når I/O er ferdig: Disk og andre I/O-enheter sender avbruddsignal til CPU når de er ferdige med en lese/skriveoperasjon, Prosessen som ventet på I/O vil da gå fra status "blocked" til "ready" og er klar til å kjøre igjen, Scheduler må velge hva som skal skje etter at avbrudds-håndteringen er ferdig: – Kjøre prosessen som ventet på I/O-enheten? – Kjøre prosessen som ble avbrutt av I/O-enheten? – Eller velge en annen prosess fra ready-køen?

To ulike strategier for scheduling: Cooperative (non-preemptive) scheduling og Preemptive scheduling. to preempt: – å ordne eller gjøre noe klart i forkant av at en hendelse skal skje,  
– å forhåndsbestille  
Cooperative (non-preemptive) scheduling: Prosessen som kjøres avbrytes ikke av OS – kan få  
kjøre "så lenge den vil", Gjør vanligvis ikke scheduling ved evt. klokkeavbrudd \* , Scheduling gjøres bare når kjørende prosess terminerer eller blokkerer, Prosessene kan samarbeide, ved at de selv gir fra seg CPU til andre prosesser  
\*: Systemer med cooperative scheduling trenger ikke å ha en timer som gjør klokkeavbrudd i hardware  
Preemptive scheduling: OS kan bestemme når scheduling skal gjøres ved å bruke klokkeavbrudd, Kjørende prosess får tildelt et maksimalt tidsintervall, Scheduling gjøres hvis prosessen fortsatt kjører etter utløp av tidsintervallet  
\*: Preemptive scheduling krever at systemet gjør klokkeavbrudd, typisk 40-50 ganger i sekundet

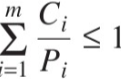
**Begreper:** Context switch: – Operasjonen i OS som utfører byttet av kjørende prosess  
eller kjørende tråd i CPU – Gjøres etter at scheduler har valgt ut neste prosess/tråd som skal kjøre – Må lagre unna alle data for nåværende prosess og deretter laste inn ny prosess som skal kjøre. En context switch er uproduktiv “overhead”, og bør være så rask som mulig for effektiv utnyttelse av maskinen  
Hovedstegene i en context switch mellom to prosesser: Scheduling-algoritmen velger neste prosess som skal få kjøre i CPU, Prosessen som kjører stoppes, alle prosessdataene lagres i områder av RAM som er reservert for bruk av OS selv, Alle data for neste prosess hentes fra RAM til CPU – her kan også CPUens cacheminne og minnehåndterings-  
enhet\* måtte oppdateres. Context switch ferdig, valgt prosess kjøres i CPU frem til  
neste context switch \*: MMU – Memory Mangement Unit

Context switch og tråder: Switch mellom tråder i samme prosess er raskere enn mellom ulike prosesser: – Trådene deler alltid kode, må ikke leses på nytt fra RAM ved switch – Trådene deler også oftest samme minneområde – Trenger bare å lagre og lese inn tråd-datene: Innhold i registre, programteller og eksekveringsstack, Brukertråder kan switches i runtime-systemet utenfor OS-kjernen, raskere enn kjernetråder

CPU-bundet og IO-bundet prosess: Alle prosesser veksler mellom "regning" i CPU ("CPU  
bursts") og venting på I/O (som foregår utenfor CPU). Prosesser som gjør mest "regning" (lange "bursts") vil ha en hastighet som avhenger mest av hvor rask. CPU – de er bundet til CPU ("CPU bound"). Prosesser som gjør mye I/O og har korte "bursts" vil være bundet til hastigheten på I/O ("I/O-bound")  
Scheduling av CPU- og IO-bundete prosesser: Etterhvert som vi får stadig raskere CPUer blir flere prosesser IO-bundet (hvorfor?) Scheduleren kan ha informasjon om en prosess er CPU-  
eller I/O-bundet. Prosesser som er I/O-bundet bør (hvis mulig) prioriteres av scheduler slik at den (relativt) langsomme disken jobber så mye som mulig parallelt med CPU. Merk at det kan være vanskelig å utnytte CPU godt hvis de fleste prosessene i stor grad er I/O-bundet

Tre ulike kategorier av datamaskinsystemer: **Batch-systemer:** Prossene som venter på å kjøres ligger i en ready-kø, Hovedregelen er at hver prosess startes og får kjøre inntil den er ferdig, Det gjøres scheduling (oftest) bare når kjørende prosess blokkerer eller er ferdig, Brukes når det er mange og ofte store/tunge jobber som kan kjøres i rekkefølge, Typisk store mengder businesstransaksjoner eller tunge beregninger som bør kjøres så raskt som mulig, Lite eller ingen interaktivitet med brukere   
Scheduling Algorithm Goals in Batch Systems: Throughput - maximize jobs per hour, Turnaround time - minimize time between submission and termination, CPU utilization - keep the CPU busy all the time  
Scheduling Algorithms in Batch Systems: First-Come First-Served i batch-systemer: "Førstemann til mølla": Prossene kjøres alltid i samme rekkfølge som de ankommer ready-køen, Når en prosess blokkerer, settes den bakerst i køen når den blir ready igjen. Fordeler: – Raskt, tar alltid bare ut første prosess i køen – Enkelt å forstå og implementere  
Ulemper: – Lite effektivt og "urettferdig" hvis mange I/O-bundete prosesser og noen få CPU-bundete prosesser  
Shortest Job First i batch-systemer: Anta at for alle prosessene i ready-kø er (ca.) total kjøretid for hele jobben som prosessen skal gjøre kjent, Scheduleren velger alltid å kjøre prosessen som har kortest total kjøretid, Vil gi kortere gjennomsnittlig ventetid i kø og bedre "turnaround" i batchsystemer enn First-Come First-Served, Variant: "Shortest Remaining Time Next" – Velg alltid prosessen som har kortest gjenværende tid – Får nye korte "små-jobber" til å bli ferdig raskere  
Shortest Remaining Time Next.  
**Interaktive systemer:** Vanligste typen datamaskinsystemer: – Standard PCer som kjører Linux/Windows/MacOS med én interaktiv pålogget bruker – Systemer med flere interaktive brukere logget på samtidig, f.eks. HiØs Linuxserver student.hiof.no – Webservere, databaseservere etc. med interaktive brukere som sender requests over nettverksforbindelser  
Scheduleren må samtidig kunne håndtere: – Interaktive forgrunnsprosesser som kommuniserer med brukere, noen av dem kanskje med real-time krav(!) – Bakgrunnsprosesser som f.eks. mail-daemon og spooler.   
Scheduling Algorithm Goals in Interactive Systems: Response time - respond to requests quickly, Proportionality - meet users' expectations  
Round-Robin scheduling: "First-In First-Out med klokkeavbrudd", Nye prosesser settes bakerst i ready-kø, Scheduler velger alltid første prosess i køen, som tas ut og får kjøre i CPU, Kjørende prosess tildeles et tidsintervall – "time slice" eller "quantum" – som er tiden den maksimalt kan få kjøre i CPU, Ved utløp av tidsintervallet gjøres klokkeavbrudd, prosessen tas ut og settes igjen bakerst i ready-køen, Scheduling gjøres (selvfølgelig) også hvis kjørende prosess terminerer eller blokkeres  
Egenskaper ved Round-Robin: "Rettferdig" og enkel å forstå/implementere, Effektiviteten avhenger av lengden på time slice/quantum. Kort time slice (< 10 ms): – Rask interaktiv respons – Mange context switch (> 100 per sekund), mye "tomgang" – Dårligere CPU-utnyttelse. Lang time slice (> 100 ms): – Utnytter CPU bedre – Treg interaktiv respons hvis mange prosesser. Vanlig å velge en "trade-off": ~40-50 ms time slice  
**Regning med Round-Robin (1):** n prosesser kjøres på én CPU med Round-Robin scheduling  
Alle prosessene er CPU-bundete og gjør ikke I/O. Vi kjenner: – Prosessenes ankomst-tid – når de settes inn i kø – Hvor mye CPU-tid hver prosess krever før den er ferdig – Systemets time slice (quantum)  
Beregn og sett opp en tabell som for hver prosess viser: – Ferdig-tid: Når prosessen er ferdig og tas ut av ready-køen – Turnaround: Tiden fra prosessen ankom til den var helt ferdig   
– Ventetid: Hvor lenge prosessen ventet i kø uten å kjøre  
**2:** n = 4, time slice 2 tidsenheter  
Antar ingen overhead (context switch skjer umiddelbart)  
Antar at nye prosesser settes i kø før kjørende prosess tas ut og evt. settes inn igjen i kø  
**3:** Trenger å finne ut når hver prosess er ferdig. Kan løses ved å "eksekvere" scheduling manuelt et tidssteg om gangen. Gjør f.eks. beregning/telling med en tabell som for hvert tidssteg viser: – Hvilken prosess som nå skal kjøre i CPU – Prosessene som er "ready" i kø   
– Hvor mye CPU-tid hver prosess har brukt til nå – Hvilke prosesser som er ferdige  
**5:** Turnaround = Ferdig-tid – Ankomst-tid Ventetid = Turnaround – CPU-tid  
**Simulering av Round-Robin:** C-program som bruker en array-kø for prosessene. Leser time-slice, antall prosesser, ankomst-tider og totalt. CPU-forbruk fra standard input (redirigeres til fil). Simuleringsløkke som kjører en time-slice (eller inntil prosess terminerer) i hvert gjennomløp. Legger ankommende prosesser i kø etter hvert som systemtiden tikker fremover.  
Skriver ut resultater: Ferdig-tider, turn-around og ventetider  
**Priority Scheduling:** Round-Robin med vanlig ready-kø: – Prosesser settes inn bakerst – Må vente på å få kjøre til de er kommer først i køen  
Scheduling med prioritetskø: – Alle prosesser i ready-kø har en prioritet (et positivt tall)   
– Scheduler velger alltid prosessen med høyest prioritet som den neste som får kjøre  
Prosessene får prioriteter for å forbedre ytelse og/eller få til en bestemt systemoppførsel  
Eksempler: – Prosesser som skal respondere i real-time, f.eks. visning av video eller håndtering av kamera/mikrofon, må få høy prioritet – Interaktive prosesser får høyere prioritet enn daemons, for å få rask responstid – I/O-bundete prosesser får høy prioritet for å utnytte hele systemet bedre – Kunder som har betalt ekstra får kjøre prosessene raskere – Forelesere bør få mer CPU-tid på HiØs server enn studenter >:-)  
Men... da får bare én prosess kjøre? Hvis prioritetene er statiske og ikke forandres underveis:  
– Scheduler velger hele tiden den samme prosessen med høyest prioritet i køen! – Ingen andre får bruke CPU før "sjefsprosessen" er ferdig  
Kan løses på flere måter: – Sett en øvre grense for hvor lenge en prosess kan bruke CPU før neste (med nest høyest prioritet) får overta – Sette en nedre grense for hvor lenge prosesser kan være i kø uten å få kjøre – Bruk dynamiske prioriteter som forandres etterhvert som prosessene kjører, slik at alle får mulighet til å bruke CPU   
**Dynamiske prioriteter:** Vil unngå "starvation" og sikre at alle prosesser får kjøre i CPU  
En mulig løsning: – Prioriteten til kjørende prosess kan reduseres litt hver gang den har kjørt i en time slice – Prioritetene blir da mindre etter hvert som prosessenes forbruk av CPU-tid vokser – Ingen CPU-bundete prosesser får "eie" prosessoren i lange perioder  
Prioritering av I/O-bundete prosesser (1): I/O-bundete prosesser gjør typisk et kort CPU-burst og blokkerer deretter for I/O. I/O går parallelt med prosessoren(e) og bruker ikke CPU-tid. I/O-operasjoner bør derfor kjøre så ofte og så mye som mulig for å utnytte systemet godt og redusere turnaround-tider. En I/O-bundet prosess som er i ready-tilstand bør få kjøre igjen så raskt som mulig for å sette i gang mer I/O. Hvis de I/O-bundete prosessene henger lenge i systemet utnyttes også RAM dårlig  
**2:** I/O-bundete prosesser blokkerer ofte for I/O før hele time slice i CPU er brukt opp. La f være andelen av time slice som en prosess brukte forrige gang den var i CPU, f.eks.:   
– Prosessen kjørte i 5 av 50 ms før den blokkerte for I/O – f = 5/50 = 0.1  
La prioriteten til hver prosess være 1/f : – Eksemplet ovenfor får prioritet lik 1/(0.1) = 10 – En CPU-bundet prosess som brukte hele timeslice får prioritet lik 1 – Korte CPU-bursts gir høy prioritet  
**Scheduling med dynamiske prioriteter i Linux:** Bruker algoritmen “Completely Fair Scheduler“ (CFS), CFS prøver å utnytte en eller flere CPU'er så effektivt som mulig og samtidig prioritere interaktive prosesser  
Hovedregel for scheduling: Velg prosessen som har brukt minst CPU-tid til nå  
Interaktivitet prioriteres av CFS: – Interaktive prosesser er mye blokkerte og "sover" fordi  
de venter på input fra bruker – De forbruker derfor lite CPU-tid – CFS gir både "sleeper fairness" og prioriterer I/O-bundete prosesser med korte CPU-bursts  
**Guaranteed Scheduling:** Setter et kvantitativt mål for scheduling, f.eks: a) Alle n prosesser skal få like mye av CPU-tiden som brukes b) Eller alle brukere skal få like stor andel av CPU  
Bruker prioriteteter som bidrar til at dette målet oppnås: Ta vare på CPU-tiden t\_used(p) hver prosess p har fått bruke. Beregn CPU-tiden t\_target(p) som hver prosess skulle hatt hvis fordelingen var "rettferdig". Velg alltid den prosessen p som er lengst unna målet, dvs. den som har den minste verdien av: t\_used(p) / t\_target(p)  
**Lottery Scheduling:** Hver prosess i ready-køen får tildelt et eller flere "lodd"/ "lottery tickets" (vanligvis bare en sekvens med heltall). Scheduling gjøres ved tilfeldig trekking av et av loddene, der alle lodd har like stor sannsynlighet for å bli trukket. Prosesser med høy prioritet får flere lodd og "vinner" ofte. Fungerer i praksis fint for å gjøre prioritering av prosesser enkelt og effektivt. Kan bidra til god respons: Nye prosesser startes tidlig. Prosessene kan samarbeide ved å utveksle lodd, f.eks. kan klient som blokkerer for I/O gi loddene til server  
**Fair-Share Scheduling:** Hver bruker skal ha en bestemt prosentandel CPU-tid. I stedet for å bruke prioriteter på prosessene, ser scheduler på hvilken bruker som eier hver prosess. Det velges en rekkefølge av prosessene som sikrer at hver bruker får sin "fair share" av CPU-tiden  
Bruker 1 eier prosessene A B C D, bruker 2 eier prosess E: – Begge brukere skal ha 50% av CPU-tiden: A E B E C E D E A E B E C E D E ...  
– Bruker 1 skal ha dobbelt så mye CPU-tid som bruker 2: A B E C D E A B E C D E ...

  
**Real-time systemer:** Vi bruker ulike strategier, målsettinger og algoritmer for scheduling for hver av disse tre system-kategoriene  
Scheduling Algorithm Goals for all Three System Categories: Policy enforcement - seeing that stated policy for scheduling is carried out. Fairness - giving each process a fair share of the CPU. Balance - keeping all parts of the system busy  
Datamaskinen er koblet til en eller flere eksterne fysiske enheter som f.eks. sensorer, kamera, mikrofon eller mediaspillere. Systemet må håndtere signaler til/fra enhetene svært raskt og innenfor gitte, korte tidsfrister – i sanntid / "real time". Hvis tidsfristene ikke overholdes kan det få fatale/alvorlige konsekvenser ("hard real time"), eller føre til uønsket men tolererbar systemoppførsel ("soft real time")  
Eksempler på hard real time: – ABS-bremser, autopilot, robotkirurgi  
Eksempler på soft real time: – Digitale opptak og avspilling av musikk, videokonferanser  
**Prosessene i et real-time system:** For å få til respons i sanntid, deles programvaren opp i prosesser som hver for seg håndterer en av et antall hendelser (signaler fra fysisk enhet) som kan skje i systemet. Oppførsel og CPU-tid for hver prosess er kjent og forutsigbar. Real-time prosesser har svært kort turnover, typisk maksimalt noen tidels sekunder. Når systemet skal forholde seg til hendelser, er det schedulerens oppgave å kjøre prosessene som håndterer hendelsen slik at alle tidsfrister overholdes  
Håndtering av hver hendelse i systemet krever en kjent prosesseringstid  
● Ikke-periodiske hendelser: – Forekommer uregelmessig og uforutsigbart  
● Periodiske hendelser: – Skjer med jevne mellomrom (med fast frekvens) – Real-time systemer må ofte håndtere flere strømmer med periodiske hendelser som har ulike frekvenser – Mange strømmer med høyfrekvente hendelser kan gi en total prosesstid som gjør sanntidsrespons umulig  
**Periodiske hendelser og sanntid:** For at vi skal oppleve "sanntid", må systemet rekke å bli  
ferdig med å håndtere en periodisk hendelse før neste forekomst av hendelsen  
Hvis det bare er én periodisk hendelse\*: – CPU-tiden C1 for å håndtere hendelsen må være kortere enn hendelsens periode P1 (tiden mellom hver forekomst):   
  
  
– Hvis ikke vil systemet "lagge" og bli mer og mer forsinket i hendelsehåndteringen – ikke lenger et real-time system!  
\*: Og vi også antar at context-switchingen går så raskt at vi kan se bor fra tiden det tar  
**Håndtering av flere periodiske hendelser:** Anta at det er m periodiske hendelser og at hendelse nummer i har periode Pi. Prosessen som håndterer hendelse i krever en CPU-tid lik Ci. Det er da relativt enkelt å bevise at vi ikke klarer å håndtere hendelsene i sanntid hvis summen av alle de m forholdene mellom periode og CPU-tid er større enn 1:

  
Hvis denne summen er større enn 1, betyr det alle hendelsene krever mer enn ett sekund CPU-tid per klokke-sekund! Systemet er da ikke "schedulable" og kan ikke implementeres.  
**Tall-eksempel:** Anta tre periodiske hendelser med perioder: P1 = 100 ms P2 = 200 ms P3 = 500 ms. CPU-tid for håndtering av hver hendelse: C1 = 50 ms C2 = 30 ms C3 = 100 ms  
Systemet er "schedulable" fordi: Anta at vi legger til en fjerde periodisk hendelse med periode  
P4 = 1s = 1000 ms  
Systemet vil da ikke lenger være "schedulable" hvis CPU-tiden: C4 > 150 ms

**Scheduling av tråder:** To nivåer av parallellitet i multi-thread systemer: – Prosesser som kjører parallelt med andre prosesser – Tråder som kjører parallelt i en og samme prosess  
Scheduling utføres ulikt for de to hovedtypene av tråder: – **Brukertråder,** som håndteres av en scheduler internt i run-time systemet for trådbiblioteket: Gjøres uten klokkeavbrudd, ingen time-slicing. Tråden får kjøre inntil en av disse hendelsene inntreffer: – Den er ferdig   
– Blokkerer for f.eks. I/O (hele prosessen blokkeres!) – Gir fra seg CPU frivillig til en annen tråd – Overskrider en evt. gitt maks.grense for CPU-tid.   
Scheduling i run-time med Round-Robin eller prioritetskø. Bytte av kjørende tråd er "lynraskt", ingen context switch. Kan "skreddersys" til å løse gitte problemer mer effektivt  
– **Kjernetråder**, som håndteres av OS-ets egen scheduler: Bytte av kjørende tråd er størrelseordener langsommere enn for brukertråder, fordi bytte av kjernetråd vil medføre fullt context switch med flushing av cache++  
Vanligvis samme algoritme som gjør scheduling av prosesser og tråder samtidig, med bare én ready-kø. I Linux er alt prosesser, men trådene er en spesiell type "lettvektsprosess". Tråder og prosesser behandles likt, med noen unntak. Det kan f.eks. gis prioritet til tråder som tilhører samme prosess som kjørende tråd, fordi context switch da er litt raskere  
**Forskjell på scheduling av bruker- og kjernetråder:** OS kjenner ikke til brukertråder og kan bare gjøre scheduling i kjernen av prosessen som brukertrådene tilhører  
Det er derfor kun brukertråder i en og samme prosess som kan kjøre i time-slice/quantum perioden (på f.eks. 50 ms) som kjernen tildeler prosessen i CPU. Med kjernetråder kan OS gjøre scheduling av alle tråder når som helst, uavhengig av hvilken prosess de tilhører

**Algorithm – Comment:**

**Optimal** - Not implementable, but useful as a benchmark, evicts the page that will be referenced furthest in the future. Unfortunately, there is no way to determine which page this is.

**NRU (Not Recently Used)** - Very crude approximation of LRU, divides pages into four classes depending on the state of the R and M bits. A random page from the lowest-numbered class is chosen.

**FIFO (First-In, First-Out)** - Might throw out important pages, bad choice. Keeps track of the order in which pages were loaded into memory by keeping them in a linked list. Removing the oldest page then becomes trivial

**Second chance** - Big improvement over FIFO, checks if a page is in use before removing it. If it is, the page is spared.

**Clock** – Realistic, simply a different implementation of second chance. It has the same performance properties, but takes a little less time to execute the algorithm.

**LRU (Least Recently Used)** - Excellent, but difficult to implement exactly

**NFU (Not Frequently Used)** - Fairly crude approximation to LRU, It is not very good.

**Aging** - Efficient algorithm that approximates LRU well. It is a good choice.

**Working set** - Somewhat expensive to implement, gives reasonable performance

**WSClock** - Good efficient algorithm  
All in all, the two best algorithms are aging and WSClock. They are based on LRU and the working set, respectively. Both give good paging performance and can be implemented efficiently. A few other good algorithms exist, but these two are probably the most important in practice

More information about MULTICS can be found at [www.multicians.org](http://www.multicians.org).

Minnehierarkiet: Typiske hastigheter og lagringskapasitet:  
Minnetype Hastighet Kapasitet  
Registre < 1 ns ≈ 100 B  
L1 cache (på CPU) ≈ 1 ns ≈ 10 KB  
L2/L3 cache 2 - 10 ns ≈ 1 MB  
Hovedminne (RAM) 20 - 100 ns ≈ 1 GB  
Solid-State/Flash disk 100 ns - 1 μs ≈ 1 TB  
Roterende harddisk 1 ms ≈ 1 TB

Måleenheter for hastighet og kapasitet Hastighet  
ns nanosekund 10-9 ett milliarddels sekund  
μs mikrosekund 10-6 ett milliondels sekund  
ms millisekund 10-3 ett tusendels sekund  
Lagringskapasitet  
B Byte 8 bits  
KB Kilobyte 1024 B ≈ 103 B = Ett tusen bytes  
MB Megabyte 1024 KB ≈ 106 B = En million bytes  
GB Gigabyte 1024 MB ≈ 109 B = En milliard bytes  
TB Terabyte 1024 GB ≈ 1012 B = En billion bytes

Hovedminne/RAM\*: Adresserbare lagerceller, RAM er bygd opp av minneceller, Hver minnecelle lagrer oftest 8 bits – en byte, Hver minnecelle har en unik adresse (et heltall) som kan brukes for å lese cellens verdi eller eller lagre en verdi. Lesing og skriving skjer ved å overføre data på databussen, mellom minnecellen og et register i CPU. Minnecellene ligger i minnebrikker (memory banks) på hovedkortet i maskinen  
\*: Aka. internminne, primærminne, main memory

Minne/RAM er en begrenset ressurs: Parkinson's lov: – "Work expands so as to fill the time available for its completion"  
Det samme gjelder for RAM: – "Programs expand to fill the memory available to hold them"  
Fra læreboken: – "An average home computer nowadays has 10,000 times more memory than the IBM 7094, the largest computer in the world in the early 1960s" ... men allikevel får vi "aldri" nok RAM!  
Hva skjer når vi slipper opp for minne? Hvis alle prosessene som kjører tilsammen trenger mer minne enn vi har på systemet: – Innholdet i minnet for en eller flere prosesser må lagres midlertidig på disk – "memory swapping"  
Diskaksesser er størrelsesordener tregere enn lesing/skriving av RAM, Systemet kan derfor bli veldig mye tregere når RAM er fullt, Vi kan risikere "trashing" – minne må hele tiden leses fra og skrives til disk og systemet blir til slutt ubrukelig

Hvorfor er minnehåndtering viktig? Alle prosesser og tråder trenger å bruke minnet, OS-kjernen har også all kode og interne data lagret i RAM så lenge systemet kjører  
Uten god minnehåndtering virker ikke maskinen: – Feil i minnet gir bugs som er “umulige” å finne for programmerere – Dårlig minnehåndtering gir en treg maskin – Usikre minnesystemer åpner for virus og malware

En av de viktigst jobbene til OS er å håndtere maskinens RAM. Programvare i OS – en Memory Manager – håndterer hovedminnet som brukes av alle prosessene. Maskinvare for minneaddressering internt i CPU – en Memory Management Unit (MMU) – sørger for korrekt  
lesing/skriving i RAM over data- og adressebuss  
I tillegg holder run-time systemer (f.eks. JVM og pthreads) rede på variablene i kjørende programmer som er skrevet i høynivå programmeringsspråk

Hovedoppgaver for Memory Manager i OS: Holde rede på status – ledig, opptatt, prosess-/tråd-tilhørighet – for de ulike delene av hovedminnet, Tildele og reservere RAM til prosessene når de starter, Finne “passende deler” av minnet ved tildeling, Evt. tildele mer minne til en prosess mens den kjører, Frigjøre og "gjenvinne" minne som ikke lenger er i bruk, Beskyttelse og deling av minne mellom prosesser

På 1950 brukte programmererne de fysiske minne-adressene i lavnivå programkode  
Høynivå programmeringspråk\* innførte en abstraksjon: – Programmereren håndterte minnet gjennom variabler – En kompilator oversatte programkoden til håndtering av minneadresser med lavnivå maskininstruksjoner – OS håndterte fortsatt fysiske minneadresser  
Etter hvert som datamaskinene ble mer kompliserte, ble det utviklet abstrakte modeller for minnehåndtering. Moderne OS har kompliserte systemer for effektiv minnehåndtering som implementerer slike modeller  
\*: IBM var først ute med høynivå-språket FORTRAN, ca. 1955

Minnehåndtering uten abstraksjon: RAM er en "lang array" av minneceller (på 8 bits = 1 byte), Hver minnecelle har en unik adresse (indeks i arrayen). I tillegg har maskinen litt Read-Only-Memory \*  
Maskininstruksjoner i CPU bruker direkte fysiske adresser: MOV REGISTER1,1000 Flytt innhold i minnecelle 1000 til register 1 i CPU JMP 208 Les neste instruksjon fra celle 208  
("hopp" til et annet sted i koden)  
Fysiske adresser er raskt og effektivt, men usikkert – ingen beskyttelse av minnet. OS er heller ikke beskyttet hvis det ligger i RAM  
\*: ROM aka BIOS – Basic Input Output System  
En prosess kan tildeles en sammenhengende del av minnet. Andre prosesser skal da i utgangspunktet ikke ha tilgang til denne prosessens minneområde.  
Ingen abstraksjon: Én prosess om gangen. Prosesser kan skrive "hvor som helst" i minnet uten  
abstraksjon eller noen form for beskyttelse. Derfor oftest bare én prosess om gangen, i tillegg til OS: – Bruker gir en kommando til OS – OS leser inn tilsvarende program fra disk til RAM – Prosessen kjøres til den er ferdig og neste kommando kan gis – Minnet skrives da over med innholdet for neste prosess  
Kan få til "multiprogramming" ved å swappe hele minnet for prosessene til og fra disk mellom hver "context switch"

En enkel abstraksjon: Adresserom  
Hvis prosessene har direkte tilgang til det fysiske minnet: – Lett å ødelegge både for andre prosesser og OS – Vanskelig å få til concurrency og multiprogramming  
En bedre løsning: Hver prosess får sitt eget adresserom: – Prosessen får tildelt en mengde abstrakte minneadresser – F.eks. indekser fra 0 opp til en maks.verdi – Adressene i en prosess er uavhengige av andre prosesser – OS sikrer at minnetilgang i adresserommet "oversettes" til korrekt og trygg tilgang til det fysiske minnet  
Adresserom med base/limit\* registere i CPU: Alle prosesser forholder seg til hvert sitt sammenhengende minneområde. Koden for prosessen bruker et minneområde med et abstrakt  
adresserom som går fra 0 opp til en maks. adresse. Prosessen tildeles et fysisk minneområde av OS. For hver prosess lagrer OS: – Base: Adressen til starten på fysisk minneområde i RAM  
– Limit: Størrelsen/lengden på fysisk minneområde  
Når prosessen skal kjøres, legges base og limit i egne registre i CPU  
Bruk av base/limit registere: Minnehåndteringen foregår i hardware i CPU. Hver gang prosessen skal lese eller skrive i minnet: – CPU sjekker at minneadressen er mindre enn limit – CPU legger til base på minneadressen  
De to registrene (bør) beskyttes, slik at bare OS får endre innholdet av dem og systemfeil i minnet unngås  
Løsningen er ikke vanlig å bruke lenger: – For mye overhead til addisjon av base for hver minneaksess – Dårlig løsning når det er mange prosesser som ikke alle får  
plass i RAM samtidig

Hva hvis alle prosesser ikke får plass i RAM? I en moderne PC vil ofte mange multimedia-applikasjoner og annen minnekrevende programvare kjøre samtidig, Det er ikke plass i RAM til alle prosessene i ready-køen, Hvis hver prosess har ett sammenhengende minne-område, kan problemet løses med swapping: – Minnet til en prosess som ikke kjører (ofte) lagres på disk – Leses fra disk og legges tilbake i RAM ved neste kjøring – Krever dynamisk relokasjon av minnet, f.eks. med base/limit registre

Håndtering av ledige/opptatte minneområder: Dynamisk tildeling og frigivelse av minne gir et "hullete" RAM med ledige og opptatte områder av ulike størrelser. OS trenger oversikt over alle disse områdene for å kunne gjøre effektiv tildeling av ledig minne til nye prosesser  
Minnet deles inn små deler allokeringsenheter som er det minste minneområdet som kan tildeles\*. OS må lagre status for alle allokeringsenhetene i hele RAM. To mye brukte teknikker: – Bitmaps – Lenkede lister  
\*: Allokeringsenheten kan variere i størrelse mellom ulike systemer og OS, fra noen bytes til flere kilobytes

Fragmentering av minnet: Dynamisk tildeling av sammenhengende minneområder vil etterhvert resultert i et "hullete" minne som består av opptatte og ledige områder i tilfeldig rekkefølge, Ofte blir det igjen små "fragmenter" med ledig minne som ligger i mellom områder som er opptatt av prosesser, Fragmentert minne kan "repareres" ved gjøre en  
kompaktering/defragmentering av hele RAM, men da stopper systemet i mange sekunder. Sammenslåing av minneområder til større "hull" når en prosess terminerer, vil redusere fragmenteringsproblemet, Sammenslåing "ordner seg selv" med bitmaps, men må gjøres av Memory Manager i OS hvis det brukes lister

Tildeling av minne til en prosess: Når en prosess opprettes har den et krav til hvor mye minne  
som skal reserveres for prosessen  
Memory Manager i OS må da: – Søke gjennom datastrukturen for ledig/opptatt minne (bitmap  
eller liste) for å finne et ledig "hull" som har nok plass – Tildele minneområde til prosessen og markere det som “opptatt” slik at det beskyttes – Resten av "hullet" markeres som fortsatt ledig  
Hele minneområdet må frigis ved terminering av prosessen for å unngå minnelekkasje

First fit: OS tildeler første "hull" den finner som er stort nok. "Hullet" deles i to deler:   
– Prosessens minneområde markeres opptatt – Resten av hullet beholdes som ledig  
Next fit: Som first fit, men søk etter neste "hull" fortsetter alltid fra der hvor forrige søk stoppet. Strategien er rask, fordi den søker så lite som mulig  
Best fit: Søker gjennom hele minnet og tildeler det "hullet" som passer best – dvs. det minste av de som har nok plass. Deler "hullet" som finnes i a) en stor del til prosessen og b) en liten, ledig del som blir igjen. Lager etterhvert mange små "hull" i minnet som ikke kan brukes til noe ... best fit "sløser" derfor mer med minne enn first fit(!)  
Worst fit er oftest bedre: Velg alltid det største "hullet" i RAM, slik at det ledige området som blir igjen forhåpentligvis er stort nok til å kunne brukes til andre prosesser  
"Quick fit" tildeling av minne: OS vedlikeholder flere ulike lister, hver av dem med ledige  
områder/"hull" av en bestemt og ofte brukt størrelse. Pluss en liste med "uvanlige" størrelser. Når en prosess ber om en "standardstørrelse" plukkes bare ett område fra listen med riktig størrelse. Raskt å tildele, ingen søking nødvendig. Krever mer arbeid for å vedlikeholde og slå sammen til større "hull" når en prosess terminerer

Bloatware og RAM-kapasitet:  
bloatware – noun – computing: Programs that use a lot  
of space and memory, often programs that are already  
on a computer or phone when it is bought, and that may  
not be needed by the user. – The Cambridge English Dictionary  
PC'ene våre får stadig flere svært RAM-krevende prosesser, spesielt multimedia-applikasjoner som kan kreve flere GB. Noen av prosessene er i seg selv for store til å passe inn i RAM. Det samlede RAM-behovet til alle prosesser på en PC vil "alltid" være mer enn det som er tilgjengelig. Trenger å kunne kjøre programmer som bare har deler av minnet sitt liggende i RAM

Sammenhengende minneområder og swapping: Hvis hver prosess skal ha ett stort, sammenhengende fysisk minneområde, blir det etterhvert vanskelig/umulig å finne "hull" i minnet som er store nok til bloatware o.l. Det tar også alt for lang tid (flere sekunder) å swappe et helt minneområde på f.eks. noen GB til/fra disk når RAM er fullt. Det er derfor utviklet løsninger som deler opp minnet for en prosess i mindre deler, slik at noen deler kan ligge i RAM og andre midlertidig på disk: – Virtuelt minne med paging – Segmentering

Virtuelt\* minne: Hver prosess tildeles et sammenhengende tenkt (virtuelt) minneområde en tabell med minne som kan indekseres fra start til slutt. Prosessen aksesserer bare sitt eget minne ved å bruke virtuelle minneadresser. OS/CPU håndterer virtuelle minneadresser og tilordner dem til korrekte adresser i det fysiske primærminnet. Rask maskinvare – Memory Management Unit (MMU) – gjør automatisk oversettelse fra virtuell til fysisk minneaddresse  
\* virtual (in computing): not physically existing but made by software to appear to do so

Virtuelle minneområder deles inn i pages:  
Page: En sammenhengende blokk med virtuelt minne \*  
Det fysiske primærminnet deles opp i page frames et område av RAM som kan lagre det faktiske inneholdet i én enkelt virtuell page. Hver virtuell page tilordnes en fysisk page frame. OS-ets minnestyring sørger for at en prosess som adresserer en virtuell page får lest fra/skrevet til riktig page frame  
\*: Størrelsen på en page er typisk 4K på Linux-systemer  
Pagetabeller: Koblingen mellom virtuelt og fysisk minne gjøres med bruk av en eller flere pagetabeller. Tabellene vedlikeholdes av Memory Manager i OS. Hvert element i tabellen –page table entry – lagrer informasjon for én virtuell page med minne. Den viktigste informasjonen i et entry er den fysiske adressen til page frame der hver virtuell page er lagret. Vi kan tenke på pagetabell som en funksjon som avbilder virtuelle minneadresser over på fysiske minneadresser

Virtuelt minne og page faults: Det samlede virtuelle minnet som prosessene bruker kan  
være mye større enn kapasiteten til RAM. Innholdet i en page frame kan enten ligge i RAM eller være swappet ut på disk. Hvis en prosess adresserer en virtuell page som ikke ligger i  
RAM får vi et avbrudd en page fault: – Hvis prosessens minne er fullt, må Memory Manager velge en page frame som tas ut av RAM og lagres på disk\*: – Valg av page frame gjøres med en page replacement algoritme – Page som ble adressert leses fra disk inn i ledig page frame   
– Pagetabellen oppdateres slik at den peker til riktig page frame  
\*: Diskadressene til page frames som er swappet ut til disk lagres ikke i pagetabellen, men et annet sted i OS

Typisk innhold i et element (entry) i pagetabellen: Page frame number: Adressen til fysisk page frame i RAM. Present/absent bit: Er page i RAM, eller får vi page fault?   
Protection bit(s): Read-write? Read-only? (Execute?)  
Modified/"dirty" bit: Er page endret siden sist den ble lest fra disk?  
Referenced bit: Er page referert til (for lesing eller skriving)?  
Caching disabled bit: Kan page ligge i cacheminnet?

Rask oversettelse fra virtuell til fysisk adresse: Hver prosess har egen pagetabell for minnehåndtering. Store virtuelle adresserom krever mye plass til pagetabeller. Tabellene må derfor lagres i RAM  
Problem: – Oversettelse av adresse vil kreve et ekstra oppslag i pagetabell i RAM for hver enkelt minneaksess som en instruksjon i CPU gjør! – Virtuelt minne vil da mer enn doble tiden som brukes på minnehåndtering i forhold til å bruke bare fysiske adresser!  
Løsning: – Legg de mest brukte elementene i pagetabellen i rask hardware nære CPU – et Translation Lookaside Buffer aAka. "associative memory

TLB – Translation Lookaside Buffer: For de aller fleste prosesser er det bare lite antall pages som aksesseres mye de andre leses/skrives bare noen få ganger. F.eks. vil minnet som er aktivt i løkker brukes mye. Idé: Bruk en "cache" i MMU til lagre info. om mye brukte pages. Gjøres i egne hardwareenheter\* TLB for å kunne oversette virtuelle adresser uten å måtte slå opp i pagetabellen i RAM. Et TLB lagrer info. om f.eks. 256 mye brukte virtuelle pages. Går mye raskere enn å slå opp i RAM fordi: – Minnet i TLB er like raskt som CPU-registre – Et søk i TLB gjøres ikke sekvensielt, men parallelt i hardware  
\*: TLB kan også implementeres delvis i software, se s. 203 – 205 i læreboken

Virkemåte for TLB: MMU får en virtuell adresse (page, offset) fra CPU. MMU sjekker om page ligger i TLB (parallelt søk i hardware). Hvis page funnet ("TLB hit"): – MMU sjekker først om tilgangskode i TLB er OK – Deretter hentes page frame direkte fra TLB, slik at MMU kan oversette til fysisk minneadresse uten oppslag i RAM/pagetabell  
Hvis page ikke funnet ("TLB miss"): – MMU velger og fjerner et av elementene i TLB, "modified" bit for fjernet page legges tilbake i pagetabellen – MMU henter info. fra RAM/pagetabell om page som ikke ble funnet, og legger dette inn på plassen i TLB som nå er blitt ledig

Håndtering av store adresserom og mye minne: Anta at virtuelle minneadresser er på 32 bit og at det brukes pages på 4 KB = 212 B = 4096 Bytes. Siste 12 bit i adresse brukes da til offset i en 4 KB page. Det 20 første bitene kan brukes til å adressere en page. Trenger en page tabell med 220 ≈ 1 048 576 = 1 M entries for å dekke hele adresserommet for virtuelle pages. Med 64 bits adresser og 4 KB page vil adresserommet ha kunne ha 252 ≈ 4.5 billiarder ulike page-adresser(!). Vi kan ikke lagre pagetabeller som dekker hele adresserommet i RAM spiser opp alt for mye minne.

Mulig løsning: Flere nivåer av pagetabeller: Anta 32 bits adresser, f.eks. (heksadesimalt og binært): 0x00403004 = 0000 0000 0100 0000 0110 0000 0000 0100  
12 siste bits er offset i 4 KB page, deler de 20 første i to deler: 0000000001⏟ PT 1=1 →10 bits  
0000000110⏟ PT 2=3 → 10bits  
000000000100⏟Offset =4 → 12bits  
Tenker oss at vi deler pagetabellen opp i 210 like store deler, hver av dem med 210 = 1024 entries for en 4 KB page  
Bruker en tabell med 210 entries på "et nivå over" til å referere til hver av disse delene av pagetabellen \*  
På det øverste nivået bruker vi bare det antallet pekere/referanser som prosessen trenger, alle de andre settes til å være tomme  
\*: For enda større adresserom og RAM kan det brukes flere nivåer enn 2, se læreboken s. 207  
Flernivå pagetabeller: 32-bits med to-delt pagetabell-adresse MMU bruker PT1 for å indeksere toppnivå MMU bruker PT2 for å indeksere andre nivå, der pagetabell entries for hver page er lagret  
Hvis en prosess trenger 12 MB minne: - Trenger bare å bruke 3 indekser på toppnivå - Hver indeks peker til 1024 \* 4 KB = 4 MB minne - Bruker bare 4 tabeller av lengde 1 K hver for å dekke hele adresserommet med 1 M ulike adresser  
Page Replacement: Hvis en prosess adresserer en virtuell page som ikke ligger i  
RAM får vi et avbrudd – en page fault. Hvis prosessens minne er fullt må Memory Manager velge en page frame som skal tas ut av RAM, slik at det blir ledig plass. Hvis page frame som velges er modifisert siden den kom inn i RAM, må den lagres på disk før den overskrives. Page frame som forårsaket fault leses fra disk inn i ledig RAM. Pagetabellen oppdateres slik at den peker til riktig page frame. Valget av hvilken page frame som skal tas ut gjøres med en page replacement algoritme

Algoritmer for page replacement: Diskaksesser er langsomme, vil ha så få page faults som mulig. Hvis bare en tilfeldig page tas ut, vil vi ofte velge mye brukte pages. Disse vil raskt leses inn igjen – mye diskbruk og dårlig effektivitet. Page replacement algoritmer prøver å finne lite brukte pages. For å få til dette bruker de informasjon fra entries i pagetabellen

Det beste valget av page som skal tas ut av RAM: – Velg den som har lengst ventetid igjen før den blir lest fra eller skrevet til (referert til) av prosessen. Krever at vi for alle pages på forhånd vet hvor lenge det er fra nåværende tidspunkt til page blir referert til. Umulig å få til i praksis. Algoritmer i praktisk bruk prøver i stedet å få til "nesten- optimale" løsninger ved å gjøre (mer eller mindre) smarte valg som forhåpnetligvis gir få page faults

Algoritme: NRU – Not Recently Used: Bruker de to statusbit'ene for hver page i pagetabellen: – Referenced (R): 1 hvis page nylig\* er lest eller skrevet – Modified (M): 1 hvis page er skrevet siden den kom inn i RAM  
Deler opp opp alle pages i fire ulike klasser: – Class 0: R = 0 M = 0 not referenced not modified – Class 1: R = 0 M = 1 not referenced, modified – Class 2: R = 1 M = 0 referenced not modified – Class 3: R = 1 M = 1 referenced modified  
Velger tilfeldig en page fra den laveste, ikke-tomme klassen. Enkel å forstå og implementere, fungerer "greit"  
\*: Referenced-bit settes tilbake til 0 for alle pages med jevne mellomrom, f.eks. ved hvert klokkeavbrudd

Algoritme: First-In, First-Out: Memory Manager i OS lagrer en vanlig kø med alle pages  
som ligger i RAM  
Køen er First-In, First-Out (FIFO): – Sortert på hvor lenge en page har ligget i RAM  
Ved page fault velges page først i køen, som har ligget lengst i RAM. Page som leses inn legges bakerst i køen. Algoritmen tar ikke hensyn til hvor ofte page leses/skrives. Brukes derfor ikke, men danner utgangspunkt for andre algoritmer for page replacement  
Second-Chance: FIFO-kø med alle pages som ligger i RAM. Køen er sortert på hvor lenge en page har ligget i RAM  
Ved page fault: – Page først i køen ("eldst" i RAM) velges for replacement hvis den ikke har vært referert til, dvs. at referenced-bit R = 0 – Hvis page først i køen har vært referert til, dvs. R = 1, settes R = 0 og page settes bakerst igjen (får en "second chance") – Algoritmen prøver deretter å velge den som nå er først i kø  
Vil "kaste ut" pages som har ligget lenge i RAM uten å bli brukt. Fungerer også hvis alle pages har R = 1 (FIFO) eller R = 0

Algoritme: LRU – Least Recently Used  
Ligner på Not Recently Used (NRU): – NRU ser bare på hva som skjedde i siste "clock tick"  
– LRU ser "lenger tilbake"  
Idé: Pages som er mye brukt nylig vil antagelig også bli brukt på nytt om kort tid  
LRU-algoritmen tar ut den page i RAM som er minst brukt i den siste tiden før page fault skjedde. Det kan vises at LRU er tilnærmet optimal hvis vi tar vare på alle minneaksesser (vektet med tidspunkt) som gjøres for hver page. Blir for langsom i praksis, fordi LRU krever at vi vedlikeholder en sortert liste av alle pages hver gang minnet leses/skrives \*Det kan tenkes løsninger i hardware som får LRU til å gå raskere, men disse brukes heller ikke.

Algoritme: Aging – simulering/tilnærming til LRU: Alle referenced-bit R resettes til 0 etter hvert "clock tick". Hvis R = 1 etter et clock tick har page blitt brukt. Tar vare på verdien av R (0 eller 1) for alle pages for f.eks. de 8 siste clock ticks. Verdiene av R lagres i en "teller" for hver page, slik at de nyeste referansene (lese/skrive) til en page "teller mest" \*  
Ved page fault velges page som har lavest verdi av telleren  
Aging er en tilnærming til "perfekt LRU" fordi: – Vi ser bare et begrenset antall instruksjoner/ticks bakover – Oppløsningen er ikke på minnetilgang men på klokketikk  
\*: Derav algoritmens navn – "aging" – referanser til en page teller mindre jo eldre de blir

Definisjon: Working Set: Vi vet at prosesser har en tendens til å bruke noen få pages  
veldig mye og ofte (f.eks. i løkker) mens andre brukes lite og mer sjeldent. Et mål på dette får vi ved å se på hvor mange pages som er brukt i f.eks. de 10 siste, 100 siste og 1000 siste  
minneoperasjonene. "Working set" w(k,t) for et tidspunkt t er definert som: Mengden av alle pages som har vært brukt i de k siste tilgangene (lesing/skriving) til minnet for denne prosessen  
Det er opplagt at størrelsen (antall pages) på w(k,t) vil øke med økende k flere minneoperasjoner som involverer flere sider. For de fleste prosesser vil denne økningen flate ut raskt fordi minneaktiviteten er konsentrert om noen få pages

Algoritme: Working Set: Idé: Finn en page som ikke er i working set w(k,t) (for en gitt  
verdi av k) og ta denne ut av RAM. Det er svært tid- og ressurskrevende å bestemme nøyaktig  
hvilke pages som er innenfor og utenfor w(k,t). En god tilnærming er i stedet å se på hvor lang tid det har gått siden en page har blitt lest eller skrevet \*  
Dette krever at vi for alle pages regelmessig lagrer og oppdaterer tiden de sist ble brukt - time of last use. Ved en page fault vil algoritmen ta ut av RAM den første page den finner som har ligget ubrukt lenger enn en gitt grense τ. Grensen τ kan typisk være 100 millisekunder  
\*: Fordi w(k,t) forandrer seg lite for større k

Detaljer: Working Set algoritmen: For å få riktige tider må prosessens CPU-tid brukes som "klokke" – current virtual time er total CPU-tid brukt frem nå. Reference bit R må nullstilles i alle pages med jevne "clock tick". Ved en page fault går man gjennom hele pagetabellen:   
– Hvis en page har R = 1 oppdateres time of last use til å være lik current virtual time – Hvis en page har R = 0:  
Beregn: age = current virtual time – time of last use  
Hvis age > τ : Page er ikke i working set og tas ut av RAM  
Hvis ingen page ble tatt ut av RAM: – Ta ut den page med R = 0 som hadde største verdi av age

Algoritme: Working Set Clock: Samme kriterium for å velge page som i Working Set. Bruker en sirkulær page-liste med R-bit og time of last use som i den enkle Clock-algoritmen. Ser alltid først på page som "viseren peker på" den som vi stoppet på ved forrige page fault. Resten av algoritmen kan implementeres tilsvarende som for "vanlig Working Set". Working Set Clock er enkel å programmere, fungerer vanligvis godt og er mer effektiv enn "vanlig Working Set". For mulig varianter og flere detaljer se læreboken s. 219 – 221

Vanlig virtuelt minne (med paging) gir en prosess ett enkelt adresserom (med pages) som indekseres fra 0 til en maks.verdi. For mange problemer er det bedre å ha flere separate "områder" med virtuelt minne som er logisk skilt fra hverandre, med hvert sitt adresserom. Det virtuelle minnet består da av flere sammenhengende minneområder av ulik størrelse – minnesegmenter. En prosess kan ha vilkårlig mange slike segmenter.  
En mengde med prosesser er i en deadlock hvis hver prosess i mengden venter på en hendelse som bare kan utføres av en annen prosess i den samme mengden  
● Systemet er låst fordi alle prosessene er blokkerte og ingen av dem kan: – Kjøre – Frigi en ressurs – Vekkes opp  
● Hendelsen som en prosess venter på er oftest frigivelse av en ressurs som holdes av (er reservert for) en annen prosess

Ressurser i datamaskinsystemer: En ressurs tilbyr en tjeneste eller en omgivelse som brukes  
av en prosess som kjører på systemet  
● Eksempler på ressurser: – CPU – Skanner – Minne – Høyttaler – Harddisk – Blu-ray og CD-R brenner – Tapestasjon – Mutex\* – Printer – Semafor\* – Plotter – Betinget variabel\*  
\*: Delte ressurser i software

Aktiv ressurs: – Tilbyr en tjeneste – F.eks. CPU, printer  
● Passiv ressurs: – Systemressurs som brukes av prosesser eller aktive tjenester  
– F.eks. minne, nettverks-båndbredde  
● Eksklusiv ressurs: – Kan bare brukes av én prosess om gangen – F.eks. høyttaler, CPU  
● Delt ressurs: – Kan brukes av flere prosesser samtidig – F.eks. minne, båndbredde

Enkel/singel ressurs: – Bare en forekomst i systemet – F.eks. høyttaler  
● Multippel ressurs: – Flere forekomster i systemet – F.eks. CPU i et fler-prosessorsystem  
● "Preemptable" ressurs: – Kan tas fra en prosess uten at det oppstår feil eller får fatale konsekvenser – F.eks. kan prosesser tas ut av CPU ved klokkeavbrudd  
● "Non-preemptable" ressurs: – Prosessen vil kunne feile hvis ressursen tas vekk  
– F.eks. skriving av disk eller brenning av CD-R

Bruken av ressurser i prosesser  
● Ressursallokering: Tildeling av ressurser til prosessene  
● Allokering av ressurser må håndteres "godt" av OS'et for å oppnå rettferdighet, effektivitet og få eller ingen deadlocks  
● Typisk bruk av en ressurs: 1. Prosessen ber om å få en ressurs som tildeles av OS, og er  
blokkert inntil den får eksklusiv tilgang til ressursen 2. Prosessen bruker ressursen   
3. Prosessen frigir ressursen igjen etter bruk

Når får vi deadlock? Deadlock i en mengde prosesser (eller tråder) skjer når alle prosessene venter på en ressurs som bare kan frigis av en av de andre prosessene i samme mengde. Alle venter, ingen frigir noen ressurs og systemet er låst. Det kan bevises\* at en deadlock bare kan oppstå hvis det er fire betingelser i systemet som alle er oppfylt samtidig. Hvis bare én av disse betingelsene ikke er til stede, kan vi ikke få deadlock. Deadlock kan unngås, forebygges og/eller "repareres" ved å undersøke og håndtere disse betingelsene  
\*: Coffman, E.G. Jr

Conditions for Resource Deadlocks: 1. Mutual exclusion: Each resource is either currently assigned to exactly one process or is available. 2. Hold and wait: Processes currently holding resources that were granted earlier can request new resources. 3. No preemption: Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them. 4. Circular wait condition: There must be a circular list of two or more processes, each of which is waiting for a resource held by the next member of the chain.

Et bilde som inneholder sketch, sirkel, diagram, clip art

Automatisk generert beskrivelseModellering av deadlocks: Bruker rettede grafer med noder som enten er ressurser  
eller prosesser, og piler som beskriver allokeringen av ressurser til prosessene  
● Firkanter: Ressurser  
● Sirkler: Prosesser  
● Pil ("arc") fra en ressurs til en prosess: – Prosessen holder (har fått tildelt) ressursen  
● Pil fra en prosess til en ressurs:– Prosessen er blokkert og venter på å få ressursen  
● Hele modellen kalles en ressursallokeringsgrafEt bilde som inneholder tekst, skjermbilde, diagram, line

Automatisk generert beskrivelse

Hvordan kan vi se at det er en deadlock? En cykel i en graf er en "vei" der vi følger pilene fra en node til neste node, og til slutt kommer tilbake til noden vi startet i. Grafer kan enten være cykel-frie eller ha en eller flere cykler. Hvis det er en cykel i en ressurs-allokeringsgraf, har vi alltid en deadlock

**Strategies Used for Dealing with Deadlocks:** 1. Ignore the problem, maybe it will go away.  
2. Detection and recovery. Let deadlocks occur, detect them, and take action. 3. Dynamic avoidance by careful resource allocation. 4. Prevention, by structurally negating one of the four required conditions.

Strategi 1: Ignorer problemet: Aka. "The Ostrich Algorithm": – OS håndterer ikke deadlocks   
– Prosesser som er låst må evt. stoppes "manuelt" eller maskinen rebootes  
● Er en fornuftig løsning hvis: – Deadlocks skjer svært sjeldent – Kostnad (tap i effektivitet) for å forhindre deadlocks er relativt stor – Kan godta et enkelt og raskt system som ikke er "100% korrekt"  
Både (standard) Linux og Windows bruker denne strategien(!) F.eks. databasesystemer (som Win SQL) kan ikke gjøre dette

Strategi 2: Detektering og gjenoppretting. Prøver ikke å forhindre deadlocks. Finner (detect) i stedet deadlocks etter at de har inntruffet. Ulike "deadlock detection"-algoritmer for:   
– Systemer med én enkelt ressurs av hver type – Systemer med flere ressurser av hver type Etter at deadlock er funnet: – Gjør en eller annen form for "gjenoppretting" (recovery)   
– Prøver å løse opp problemet og få prosessene til å kjøre videre etterpå

Hvordan finne ut om systemet er i deadlock? Sett opp en datastruktur for ressursallokeringsgrafen\*, som viser hvilke prosesser som venter på og/eller holder  
ressurser. Kjør en algoritme som finner ut om det er en cykel i grafen. Systemet er i deadlock hvis algoritmen finner en cykel. Finnes flere algoritmer som kan brukes for å finne cykler. Hastighet/effektivitet av algoritmen ikke (veldig) viktig: – Små datasett: Begrenset antall prosesser og ressurser – Relativt lenge mellom hver gang OS gjør deadlock-sjekking  
\*: F.eks. med tabeller/arrayer som lagrer alle noder (prosesser og ressurser) og "pilene" mellom disse

Deadlock i systemer med flere ressurser av hver enkelt type: Anta at prosessene tilbys flere ressurser av hver type F.eks. 3 printere, 4 tapestasjoner, 2 Blue-ray brennere...  
Den enkle ressursgrafmodellen vil ikke virke det blir ikke deadlock før evt. alle ressursene av en type er oppbrukt. Finnes flere algoritmer for å finne deadlock i slike systemer, alle bygger på modeller der vi gjør forenklinger  
● Forenklinger i lærebokas algoritme: – Vi kjenner alle prosessenes ressursbehov til enhver tid – Ressursbehovene er statiske (forandres ikke underveis)

Flere ressurser av hver type: Systemdata Systemtilstanden lagres i fire tabeller:  
● Data om m ulike typer ressurser i to arrayer E og A: – E (Existing): Antallet ressurser som finnes av hver type – A (Available): Antallet ressurser som er tilgjengelige av hver type  
● Data om n prosesser i to n × m matriser C og R: – C (Current): Antallet ressurser av hver type som hver prosess nå holder (har reservert) – R (Request): Antallet ressurser av hver type som hver prosess ønsker å reservereEt bilde som inneholder tekst, skjermbilde, Font, diagram

Automatisk generert beskrivelse

Algoritme for å detektere deadlock: Alle prosesser er "umerkede"  
● En prosess "merkes" i hver steg i algoritmen: 1. Finn en umerket prosess P som kan få alle sine ønsker om ressurser (gitt i en rad i R) oppfylt av ressursene som er tilgjengelige i A  
2. Hvis en slik prosess P er funnet: Legg ressursene som P holder (gitt i en rad i C) til som nye  
tilgjengelige ressurser i A. Merk prosessen P og gå tilbake til steg 1  
3. Hvis ingen prosess ble funnet i steg 1: Alle prosessene som ikke er merket er i deadlock

Hvorfor virker algoritmen? Prosessen P som vi finner og merker i hvert steg får alle  
ressursene den trenger. Den kan derfor kjøre helt ferdig uten deadlock. Når P er ferdig blir ressursene den holdt tilgjengelige igjen (første delpunkt i steg 2 i algoritmen) slik at de kan  
gis til andre prosesser i neste runde. Når algoritmen terminerer, kan ingen av de evt. gjenværende umerkede prosessene få det de trenger deadlock

Hvor ofte skal systemet sjekkes for deadlocks?  
● Hver gang en prosess ber om ressurs? – Vil finne "alle" deadlocks – Koster for mye CPU-tid  
● Med faste mellomrom, f.eks, annethvert minutt? – Vil ikke finne deadlocks før lenge etter at de oppstår – Effektivt  
● Bare når CPU-utnyttelsen er lav? – Kan være pga. at mange prosesser er i deadlock...   
– Få prosesser som kan kjøre gir lite bruk av CPU

Possible Methods of recovery (though none are “attractive”):   
A: Preemption: En del ressurser er "preemptive" – de kan tas fra en prosess uten at det får "fatale" følger. Når deadlock oppstår, kan OS prøve å ta en preemptive ressurs fra en prosess og gi den til en annen som da kan få fortsette å kjøre. Er generelt vanskelig (umulig?) å automatisere med en algoritme, fordi situasjonen er uforutsigbar og i stor grad avhenger av typen ressurs. Gjøres oftest bare manuelt i "nødsituasjoner"

B: Rollback: Status for prosessene, inkludert hvilke ressurser de bruker, logges til fil med jevne mellomrom – checkpoints. Loggfilene lagres for et relativt langt tidsrom bakover i tid. Ved en deadlock kan man stoppe en eller flere prosesser som holder ressurser som forårsaker deadlock. Ressursene som frigis tildeles andre prosesser som får starte igjen. Prosessene som ble stoppet startes senere på nytt fra en tilstand de hadde på et tidligere tidspunkt – rollback\* – ved å bruke loggfilene  
\*: Samme type løsning brukes i databasesystemer for å kunne gjenopprette systemet når det skjer en feil

C: Killing processes: Brutalt, men ofte effektivt. Terminerer en eller flere prosesser for at andre som er i deadlock kan starte å kjøre igjen  
● Hvilke prosesser skal velges ut for å bli terminert? – En av prosessene i en cykel i ressursallokeringsgrafen? – En kjørende prosess som holder ressurser som prosesser i  
deadlock trenger?  
– En prosess som kan startes på nytt "uten varig skade"? ○ En kompilering kan avbrytes og senere gjenskapes ○ En oppdatering av en database bør ikke termineres

Strategi 3: Unngå deadlock: Deadlock oppstår når noen prosesser holder (for) mange ressurser, slik at andre aldri får de de trenger. Ressurser tildeles oftest til prosesser en og en om gangen. Ønsker å tildele ressurser slik at deadlock kan unngås: – Systemet "vet" om det er trygt å gi en prosess en ressurs – Ressursen tildeles bare dersom det ikke er utrygt. Det finnes algoritmer for ressurstildeling som løser dette problemet og i stor grad unngår å lage deadlocks. Algoritmene krever en del forhåndskunnskap om systemet for å kunne fungere

Sikre og usikre systemtilstander: Algoritmer for å unngå deadlocks sjekker om systemet er  
i sikker ("safe state") eller usikker ("unsafe state") tilstand  
● Safe state: – Det finnes en rekkefølge for tildeling av ressurser (og scheduling av prosessene) som gjør at alle prosessene garantert kan få kjøre ferdig (uten deadlock)  
● Unsafe state: – Finnes ingen tildelingsrekkefølge som gjør at alle prosessene garantert kan kjøre ferdig – Tilstanden kan derfor lede til en deadlock

Banker's\* algoritme for én type ressurs: Hver prosess har en "kreditt": – Systemet vet hvor mange ressurser en prosess maksimalt vil kunne trenge å holde (ha reservert) samtidig  
● Det totale antallet tilgjengelige ressurser kan være mindre enn summen av alle "kredittene"  
● Holder hele tiden rede på antallet ressurser som hver prosess holder og maksimalt trenger  
● Algoritmen tildeler bare en ressurs hvis tildelingen er sikker: – Sikker: Det finnes en videre sekvens av tildelinger som gjør at alle prosessene garantert kan få ressursene de trenger og  
derfor terminere korrekt  
\*: Navnet på algoritmen kommer fra bankvirksomhet: Banker bruker samme prinsipp når de låner ut penger og gir kreditt, for ikke å risikere å måtte bruke opp mer enn sine tilgjengelige ressurser.

Banker's algoritme med flere ressurser av hver enkelt type: Bruker omtrent de samme arrayene og matrisene for prosesser og ressurser som algoritmen for detektering av deadlocks i tilsvarende type systemer. Logikken er også relativt lik, men her undersøker vi om tilstanden er safe/sikker, dvs. at vi kan garantere at det ikke er deadlock. En tildeling av en ressurs godkjennes (som hovedregel) bare hvis tilstanden den leder til ikke er usikker

Banker’s Algorithm for Multiple Resources: 1. Look for a row, R, whose unmet resource needs are all smaller than or equal to A. If no such row exists, system will eventually deadlock. 2. Assume the process of row chosen requests all resources needed and finishes. Mark that process as terminated, add its resources to the A vector. 3. Repeat steps 1 and 2 until either all processes are marked terminated (safe state) or no process is left whose resource needs can be met (deadlock)

Hva bruker Banker's algoritme til? Egentlig svært lite...  
● Fungerer fint i teorien, men ikke i praksis, fordi: – Prosesser har sjelden oversikt over alle egne ressursbehov – Mengden med prosesser endres mye og ofte – Ressurstilgangen kan endre seg  
I stedet brukes tilnærminger og heuristikker basert på lignende idéer.   
En praktisk tilnærming: Two-Phase Locking: Enkel metode for å forhindre deadlocks som ligner litt på "kjøring med maks. antall ressurser" i Banker's algoritme. Implementeres vanligvis i applikasjonen og ikke i OS, Fase 1 (ikke-produktiv): – Prosessen prøver å reservere alle ressursene den trenger, én etter én – Hvis en ressurs ikke er tilgjengelig: Frigi alle ressursene som prosessen har fått, vent litt, og prøv igjen. Fase 2 (produktiv): – Kjør prosessen (som nå har alt den trenger) – Frigi alle reserverte ressurser

Strategi 4: Forhindre at deadlock kan skje: Banker's algoritmen for å unngå ressurstildelinger som kan gi deadlock er vanskelig (umulig?) å få til i praksis. Krever at vi har informasjon tilgjengelig om alle prosessenes fremtidige ressursønsker og det har vi vanligvis ikke. I stedet for å dele ut ressurser "forsiktig", kan vi i stedet prøve å forhindre at deadlocks i det hele tatt kan skje(!). Utgangspunktet er de fire nødvendige betingelsene som vi vet må være tilstede for å få en deadlock. Vil prøve å lage/designe systemet slik at minst én av disse betingelsene oftest ikke er oppfylt – "attack the condition"

Betingelse 1: Forhindre "mutual exclusion": Hvis alle ressurser kan deles av flere prosesser samtidig og ikke bare brukes av en om gangen, kan vi aldri få deadlocks. Går ikke i praksis: Noen ressurser som f.eks. printer og tapestasjoner kan ikke fysisk deles samtidig av flere. Problemet kan delvis løses med virtualisering av en fysisk ressurs slik at den kan deles, f.eks. med et spooling-system for printing. For å redusere sannsynlighet for deadlock: – Unngå å reservere ressurser hvis det ikke er helt nødvendig – Minimalisér antall prosesser som ber om tilgang til en og samme "mutually exclusive" ressurs

Betingelse 2: Forhindre "hold and wait": Hvis ingen prosesser som har fått en ressurs får lov til å vente (blokkere) for å få enda flere ressurser, kan vi ikke få deadlock. "Løsning": Prosesser tildeles alle ressurser de trenger før start\* – Ingen prosesser må noen gang vente på en ressurs  
Problemer med løsningen: – Krever at prosessene kjenner alle fremtidige behov (Banker's...) – Lite effektivt, blokkerer ressurser som andre prosesser trenger  
Mulig variant som også unngår "hold and wait": – Prosess som ber om en ressurs frigir først alle ressursene den holder – Den prøver deretter på nytt å få alle ressursene den trenger  
\*: Systemer som håndterer f.eks. telefonsamtaler fungerer på denne måten

Betingelse 3: Forhindre "no preemption": Hvis prosessene ikke tillates å holde på ressurser inntil de selv frigir dem ("no preemption"), men vi i stedet kan "tvinge" en prosess til f.eks. å gi en ressurs til en annen prosess ("preemption"), kan vi ikke få deadlocks  
● Dette er i praksis ikke mulig: – Noe kan løses med virtualisering av f.eks. printere med spooling – Men, det vil alltid være ressurser som ikke kan tas vekk før  
prosessen er ferdig med dem, f.eks.: 3D-printere, Tapestasjoner, Databasefelt som er låst for skriving/oppdatering, Prosess- eller filsystemtabeller internt i et OS

Betingelse 4: Forhindre "circular wait": Deadlock unngås hvis vi aldri får cykler i ressursallokeringsgrafen  
● Kan bevises\* at vi aldri får en sirkulær ventetilstand hvis: – Alle ressursene nummereres fra 1 til et maks. antall – Alle prosesser må be om/tildeles ressurser i stigende nummerrekkefølge  
– Ressursene må deretter frigis i avtagende nummerrekkefølge

Forhindre "circular wait" med nummerering av ressursene og krav til rekkefølge: Kan fungere for små systemer med noen få prosesser  
Går ikke i praksis med mange prosesser og mange ulike ressurser: – "Although numerically ordering the resources eliminates the problem of deadlocks, it may be impossible to find an ordering that satisfies everyone. When the resources include process-table slots, disk spooler space, locked database records, and other abstract resources, the number of potential resources and different uses may be so large that no ordering could possibly work."

Condition Approach  
Mutual exclusion Spool everything  
Hold and wait Request all resources initially  
No preemption Take resources away  
Circular wait Order resources numerically

Attribute Meaning

Protection Who can access the file and in what way  
Password Password needed to access the file  
Creator ID of the person who created the file  
Owner Current owner  
Read-only flag 0 for read/write; 1 for read only  
Hidden flag 0 for normal; 1 for do not display in listings  
System flag 0 for normal files; 1 for system file  
Archive flag 0 for has been backed up; 1 for needs to be backed up  
ASCII/binar y flag 0 for ASCII file; 1 for binary file  
Random access flag 0 for sequential access only; 1 for random access  
Temporar y flag 0 for normal; 1 for delete file on process exit  
Lock flags 0 for unlocked; nonzero for locked  
Record length Number of bytes in a record  
Key position Offset of the key within each record  
Key length Number of bytes in the key field  
Creation time Date and time the file was created  
Time of last access Date and time the file was last accessed  
Time of last change Date and time the file was last changed  
Current size Number of bytes in the file  
Maximum size Number of bytes the file may grow to

Files exist to store information and allow it to be retrieved later. Different systems provide different operations to allow storage and retrieval. Below is a discussion of the most common system calls relating to files.

1. Create. The file is created with no data. The purpose of the call is to announce that the file is coming and to set some of the attributes.

2. Delete. When the file is no longer needed, it has to be deleted to free up disk space. There is always a system call for this purpose.

3. Open. Before using a file, a process must open it. The purpose of the open call is to allow the system to fetch the attributes and list of disk addresses into main memory for rapid access on later calls.

4. Close. When all the accesses are finished, the attributes and disk addresses are no longer needed, so the file should be closed to free up internal table space. Many systems encourage this by imposing a maximum number of open files on processes. A disk is written in blocks, and closing a file forces writing of the file’s last block, even though that block may not be entirely full yet.

5. Read. Data are read from file. Usually, the bytes come from the current position. The caller must specify how many data are needed and must also provide a buffer to put them in.

6. Write. Data are written to the file again, usually at the current position. If the current position is the end of the file, the file’s size increases. If the current position is in the middle of the file, existing data are overwritten and lost forever.

7. Append. This call is a restricted form of write. It can add data only to the end of the file. Systems that provide a minimal set of system calls rarely have append, but many systems provide multiple ways of doing the same thing, and these systems sometimes have append.

8. Seek. For random-access files, a method is needed to specify from where to take the data. One common approach is a system call, seek, that repositions the file pointer to a specific place in the file. After this call has completed, data can be read from, or written to, that position.

9. Get attributes. Processes often need to read file attributes to do their work. For example, the UNIX make program is commonly used to manage software development projects consisting of many source files. When make is called, it examines the modification times of all the source and object files and arranges for the minimum number of compilations required to bring everything up to date. To do its job, it must look at the attributes, namely, the modification times.

10. Set attributes. Some of the attributes are user settable and can be changed after the file has been created. This system call makes that possible. The protection-mode information is an obvious example. Most of the flags also fall in this category.

11. Rename. It frequently happens that a user needs to change the name of an existing file. This system call makes that possible. It is not always strictly necessary, because the file can usually be copied to a new file with the new name, and the old file then deleted.

Essential Requirements for Long-Term Information Storage: 1. It must be possible to store a very large amount of information. 2. Information must survive termination of process using it. 3. Multiple processes must be able to access information concurrently.

Filsystemet: Vi trenger et system for å lagre og lese store mengder data med bruk av permanente media  
Data deles opp i enheter som logisk hører sammen – filer: – Filene er uavhengige av prosessene og kan brukes av "alle" – En vanlig PC kan typisk ha flere hundre tusen filer på disk  
Filsystemet i et OS er strukturen, reglene og mekanismene som brukes for å håndtere filene. Det finnes mange varianter av filsystemer for ulike OS og forskjellige lagringsmedia

De viktigste oppgavene til filsystemet: Håndtere både filenes innhold og metadata, Strukturere lagringsplassen som er tilgjengelig, Gi god pålitelighet, effektivitet og datasikkerhet, Tilby brukerne verktøy for å håndtere filer, Muliggjøre deling av data, Backup og redundans

Filer og hardware: Filer lagres på permanente media (aka sekundærminne), Lagrer store datamengder til lav kostnad, Ikke direkte tilgjengelig for CPU, Aksesseres i stedet gjennom spesielle minneceller for I/O som er koblet opp mot et lagringsmedium – oftest en disk, Data overføres raskt mellom RAM og disk\* i blokker som har en fast størrelse (typisk 4 kb = 4096 bytes som i Linux), Disken kan tenkes på som en "array" der disse blokkene ligger  
lagret sekvensielt på samme måte som "pages" i virtuelt minne  
\*: Ved bruk av DMA-brikker (Direct Memory Access) som flytter data direkte uten å gå om CPU

Noen typer permanente media, Sortert etter lese-/skrivehastighet: – Solid-state drive (SSD/Flash-disk uten bevegelige deler) – Magnetiske, roterende disker (hard disk/hard drives) – Optiske disker (CD, DVD, Blue-ray) – Magnetisk tape

Minnehierarkiet: Typiske hastigheter og lagringskapasitet  
Minnetype Hastighet Kapasitet  
Registre < 1 ns ≈ 100 B  
L1 cache (på CPU) ≈ 1 ns ≈ 10 KB  
L2/L3 cache 2 - 10 ns ≈ 1 MB  
Hovedminne (RAM) 20 - 100 ns ≈ 1 GB  
Solid-State/Flash disk 100 ns - 1 μs ≈ 1 TB  
Roterende harddisk 1 ms ≈ 1 TB

Det fysiske filsystemet: Fysisk er filsystemet bygget opp av maskinvare og  
lavnivå programvare/ROM: – Disker og andre permanente lagringsenheter – Elektronikk og elektromekanikk\* – Magnetiske spor og sektorer på overflaten av disker – Blokkbaserte I/O-enheter og disk-kontrollere – Tabeller med “fil-pekere” som lagrer fysiske diskadresser  
\*: F.eks. lese- og skrivehoder og rotasjonsmotorer som driver tradisjonelle magnetiske harddiske

OS og det logiske filsystemet: OS håndterer en modell av filsystemet en abstraksjon  
vekk fra hardware og det fysiske systemet\*  
● OS tilbyr et sett med brukerkommandoer for filhåndtering  
● OS "oversetter" brukerkommandoene til intern håndtering av filer og media  
● Brukere av maskinen trenger derfor ikke kjenne til detaljer om hardware og det fysiske filsystemet  
\*: På samme måte som OS implementerer abstrakte modeller for å håndtere prosessene og minnet

Filbegrepet: En fil \* er en grunnleggende lagringsenhet i et OS  
● Brukes for alle data som: – Lagres utenfor RAM (primærminnet) – På permanente media (sekundærminne)  
● En fil er en samling data (bytes) som logisk hører sammen: – Datafiler: Tekstdokument, bilde, lydfil, video, programvare... – Eksekverbar kode – Systemfiler, drivere...  
\*: Engelsk “file”, opprinnelig fra papirbaserte arkivsystemer

Filnavn: Alle filer må ha et navn for at vi skal kunne finne dem igjen og referere til dem fra applikasjoner/prosesser  
● Forskjellige OS bruker ulike regler for navngiving av filer: – Maks. antall tegn i navnet   
– Hvilke tegn som er tillatt å bruke i filnavn – Forskjell på store og små bokstaver   
– Oppdeling av filnavn i flere deler med ulik betydning\* – Tolking av filnavn med kobling mot bestemte applikasjoner  
\*: Typisk en “extension” på slutten av filnavnet, f.eks. .java, .txt, som angir filtypen/innholdet

"No-frills" byte sequence – used in Windows and Linux for all files/data  
Record sequence – used in older OS  
Sorted Tree – for fast searching in large data sets

Metadata (attributter) for filer: Informasjon om filen som OS lagrer i tillegg til datene. Hvilke metadata som lagres varierer mellom ulike OS

Lesing og skriving av filer: Filaksess  
● Sekvensiell aksess (tape): – Alltid fra begynnelsen av filen og fremover – Kan ikke "hoppe rundt" i filen, tar lang tid å "finne frem"  
● Random/tilfeldig aksess (disker): – Kan starte å lese/skrive alle steder i filen  
– Metode 1: ○ Hver read/write operasjon har en posisjon i filen  
– Metode 2: ○ Hver read/write bruker nåværende/current posisjon i filen ○ En egen "seek"-operasjon brukes til å sette nåværende posisjon før read/write  
– Linux og Windows bruker begge metode 2

Kataloger / Mapper / Directories: En katalog er en samling av (eller en liste med) filer som er  
"beslektet" eller hører logisk sammen. Brukes for å holde orden og lettere finne frem i alle filene. Kataloger/mapper finnes "egentlig ikke" fysisk, men må implementeres på en eller annen måte i filsystemet. De fleste OS har et hierarkisk katalogsystem: ○ En katalog/mappe kan inneholde nye (under)kataloger/mapper ○ Gir et hierarkisk (nivådelt) system som kan tegnes som et tre  
● Finnes også OS med bare ett nivå av kataloger : Typisk enkle systemer der OS er embedded (innkapslet) i f.eks. et kamera eller en bærbar mediespiller

Master Boot Record og diskpartisjoner: Disken deles opp i en eller flere partisjoner, hver partisjon inneholder et eget, uavhengig filsystem (File System – FS). I tillegg er det en egen boot-sektor, aka Master Boot Record – MBR, som ligger først på disken. MBR innholder bl.a. en partisjonstabell som angir diskadressene til start og slutt på hver diskpartisjon

Bruk av disken ved booting av systemet: Instruksjoner i ROM/BIOS leser Master Boot Record, laster et lite program som ligger i MBR inn i RAM og starter dette. En av partisjonene vil være merket som "aktiv" i MBR, denne partisjonen inneholder OS'et  
● Den aktive partisjonen inneholder en "boot block"\*  
MBR laster programmet i den aktive partisjonens boot block inn i RAM og starter dette. Programmet fra boot block laster inn hele OS og booting av systemet er ferdig  
\*: For å få alle partisjoner like i struktur inneholder alle en boot-block, også de som ikke lagrer et OS

Implementasjon av (lagring av) filer: OS må holde rede på hvilke diskblokker som utgjør en fil. Tildelingen i OS av diskblokker til en fil kalles fil-allokering. Måten dette implementeres på er viktig(st) for filsystemets effektivitet og oppførsel, og gir skillet mellom ulike typer FS  
● Fire ulike implementasjoner: – Sammenhengende (contiguous) fil-allokering – Allokering med lenkede lister – Allokering med lenkede lister med bruk av en tabell i RAM – i-noder

Sammenhengende filer: Fordeler og ulemper  
● Fordeler: – Enkelt å implementere – Svært effektivt: Trenger bare å lagre starten på hver fil,  
sekvensiell lesing av filen utnytter hele diskens båndbredde  
● Ulemper: – Må vite størrelsen på alle filer før de opprettes – Gir etterhvert en sterkt fragmentert disk med mange "hull" – Krever regelmessig komapktering, som er svært tidkrevende  
● Brukes bare for CD-ROM, DVD, Blu-rays og andre media som brennes/skrives bare en gang, der alle filstørrelser er kjente før skrivingen starter

Filer med lenkede lister: Fordeler og ulemper  
● Fordeler: – OS trenger bare å lagre adressen til første blokk på filen – Løser fragmenteringsproblemet, filen ligger spredt utover disk  
● Ulemper: – Ikke-sekvensiell/random tilgang til filen går svært sakte må følge en kjede av diskadresser som alle må leses fra disk for å komme til stedet i filen der lesing/skriving skal starte – Antall data som kan lagres i en blokk er ikke lenger en  
toerpotens kan gi problemer med effektivitet av blokkbaserte I/O-enheter – Ulempene er såpass alvorlige at denne løsningen ikke brukes i praksis i noe OS

Lenkede lister med "neste-pekerne" i RAM: • Løsning: “Neste-pekeren” for alle diskblokkene lagres i RAM i en tabell som OS selv vedlikeholder • Tabellen kalles en File Allocation Table (FAT), som har gitt navn til en rekke filsystemer som bruker denne løsningen bl.a. i eldre utgaver av Microsoft-OS\*  
• Løser problemene forbundet med å ha "nestepekere" på disk  
● Ulempe: – Tabell med diskadresser må lagres i RAM og kan bli svært stor – Størrelsen på FAT avhenger lineært av disk- og blokkstørrelse – F.eks. kan en disk på 1 TB (10¹²) bytes kreve en FAT med 1 milliard (10 ) elementer hvis blokkstørrelsen er 1 KB⁹  
\*: Nyere versjoner av Windows bruker filsystemet NTFS, der blokk-info. lagres i B-trær

“Slack space“: Sløsing med diskplass  
● Slack space for en fil: – Ekstra lagringsplass som brukes fordi filen ikke fyller opp  
diskblokken(e) som er satt av – Gjennomsnittlig slack space for en fil er halvparten av filsystemets blokkstørrelse  
● Store diskblokker: – Mye slack space, spesielt hvis mange små filer  
● Små diskblokker: – Mindre slack space, men mye overhead i filsystemet hvis det er mange store filer

Fragmentering av filer: Disken deles opp i blokker på f.eks. 4 KB. Filsystemet i OS holder rede på hvilke blokker som er ledige og hvilke som er i bruk  
● Fragmentering: – Når en fil blir større enn de ledige sammenhengende områdene på disk, må den lagres i områder som er spredt rundt på disken filen blir fragmentert

Fragmentering og diskhastighet  
● For hver fil må OS holde rede på: – Hvor på disken de ulike blokkene som tilsammen utgjør en fil ligger lagret – I hvilken rekkefølge disse delene av filen skal settes sammen  
● Etter lengre tids bruk kan en disk bli sterkt fragmentert: – Store filer består av svært mange småbiter som ligger tilfeldig spredt på disken – Lesing/skriving av sterkt fragmenterte filer er lite effektivt, spesielt for tradisjonelle magnetiske disker

Håndtering av fragmentering  
● Prøv å unngå fragmentering: – Legg filer som leses mye, men ikke oppdateres ofte, på en egen disk, og f.eks. loggfiler/cache etc. på en annen – Prøv å sette av “nok plass” ved opprettelse av filer som “skjøtes på” ofte. – Bruk større diskblokker  
● Defragmentering: – Gå gjennom disken med jevne mellomrom for å gjøre fragmenterte filer sammenhengende igjen – Ressurs- og tidkrevende, kan bare gjøres ved lav belastning på systemet