Operativsystemer

- Introduction
- Processes and threads
- Memory management
- File systems
- Input/Output
- Deadlocks
- Multiple processor systems
- Security
- Case study: Linux

Kapittel1: “Introduksjon”

Hva er et OS?:

Systemkall:

Systemkall som har med håndtering av prosesser:

- Fork
- Waitpid
- exevc

Priviligerte instruksjoner:

Hvordan kan en brukerprosess få tak i operativsystemets tjenester, som for eksempel for å skrive data til ei fil?:
Operativsystemet tilbyr sine tjenester enten via bibliotekskall eller ved at den har en prosess som betjener tjenesten. Brukerprosessen vil kalle en biblioteksfunksjon som igjen må gjøre et systemkall. Når systemkallet skjer blir konteksten for CPUen byttet til kjernemodus slik at operativsystemet kan beskytte den ressusen som tjenesten gjør bruk av, for eksempel for å hindre at brukerprosessen ødelegger ei fil.

Hva skjer når en prosess kjører en TRAP-instruksjon?
En TRAP-instruksjon brukes for å implementere et systemkall og det er instruksjonen som skifter fra brukermodus til kjernemodus.

Kapittel 2: “Prosesser & Tråder”

2.1: Prosesser **2.2: Tråder** **2.3: Synkronisering**	**2.4: Tidsdeling** **2.5: Klassiske syn. problemer.**

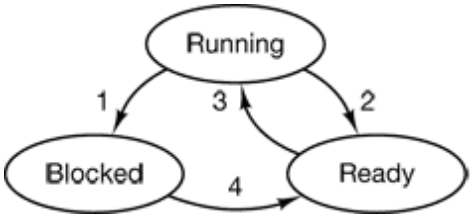
Prosesser:

Hva er en prosess? Et program i utførelse.

- Adresserom
- Programteller
- Register
- I/O

Prosesstilstander:

- **Running**
- **Blocked**
- **Ready**
- **Termination** → En prosess terminerer og er ferdig.



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Start og stopp av prosesser

- **Start:**
  1) System initialization
  2) System call by a running process (systemkall, create/fork)
  3) A user request to create a new process
  4) Initiation of a batch job (script)
- **Stop:**
  1) Normal exit (fivillig)
  2) Error exit (frivillig)
  3) Fatal error (ikke frivillig, deling på null)
  4) Killed by another process (ikke frivillig)

Prosesskifte (kontekstsvitsj):

In a switch, the state of the first process must be saved somehow, so that, when the scheduler

gets back to the execution of the first process, it can restore this state and continue.
The state of the process includes all the registers that the process may be using, especially the program counter, plus any other operating system specific data that may be necessary.

Hva trigger en kontekstsvitsj?

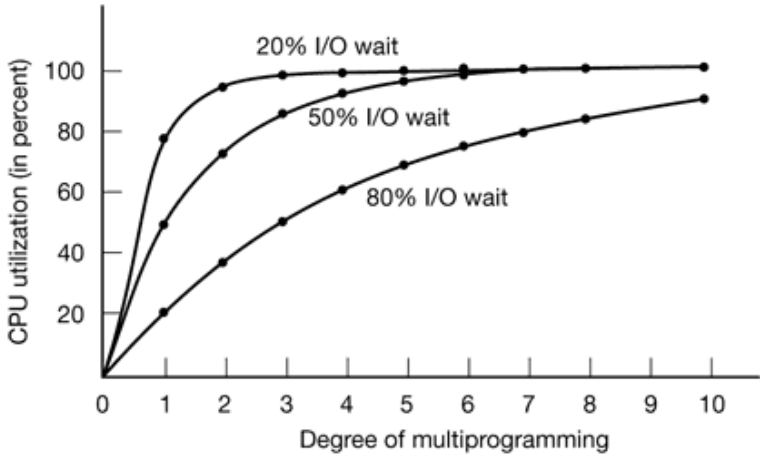
1) Multitasking (flere prosesser må dele CPU)
2) Avbrudd (interrupt)
3) Bytte mellom user/kernel space

Hva skjer ved et avbrudd (interrupt)? (mer i detalj)

1) Hardware stacks programcounter
2) Hardware loads new programcounter (from interrupt vector)
3) Assembly language procedure saves registers
4) Assembly language procedure sets up new stack
5) C interrupts service runs (typically reads and buffers input)
6) Scheduler diides which process is to run next
7) C procedure returns to the assembly code
8) Assembly language procedure starts up new current process

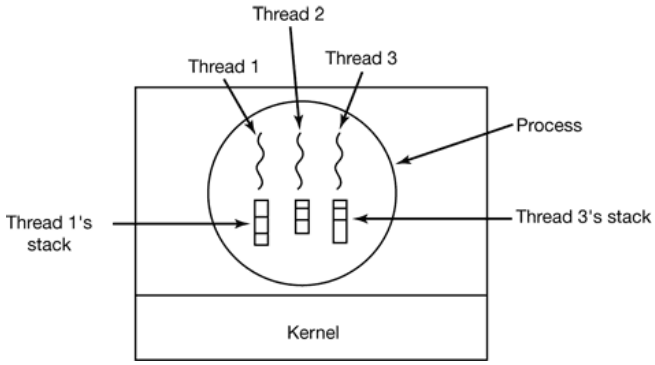
Annet:

- Hva er et interrupt
- CPU utilization = 1 - p^n
  fraction waiting for I/O = p
  Processes in memory = n



Tråder

Hva er en tråd?



Hvorfor tråder?

- Programmer har mange aktiviteter
- Paralellprogrammering er enklere med tråder
- Deling av data blir enklere (tråder kan dele data)
- Bedre ytelse og utnyttelse av multiple CPUer og cores
- Kan ha blokkerende systemkall

3måter å bygge en server

- **Threads:**
  - Paralellism
  - Blocking system calls
- **Single threaded proces:**
  - No paralellism
  - Blocking system calls
- **Finite state machine:**
  - Paralellism
  - No blocking system calls
  - Interrupts

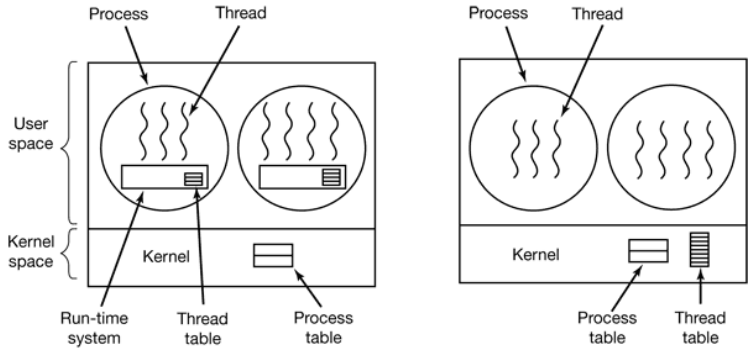
Prosess vs. tråd

Pr. process item:	**Pr. thread item**
- Address space - Global variables - Open filles - Child processes - Pending alarms - Signals and signal handlers - Accounting information	- Program counter - Registers - Stack - State

Posix threads

Kjerne- og brukertråd:
When threads are managed in user space, each process needs its own private **thread table** to keep track of the threads in that process. This table is analogous to the kernel's process table, except that it keeps track only of the per-thread properties such the each thread's program counter stack pointer, registers, state, etc. The thread table is managed by the runtime system. When a thread is moved to ready state or blocked state, the information needed to restart it is stored in the thread table, exactly the same way as the kernel stores information about processes in the process table.

Kjernetråd vs. brukertråd
- (+) Håndterer blokkerende systemkall
- (+) Håndterer sidefeil
- (+) Tillater sannparallelitet i samme prosess
- (-) Tidsdeling er bestemt av OS
- (-) Dyrt å gå inn i kjernemodus for å skifte tråd



Omprogrammering fra single-threaded til multi-threaded
- Problemer
  - Konflikt med bruk av globale variable
    → En løsning kan være at hver tråd har en kopi av de globale variablene.
  - Ikke-restartbare biblioteksrutiner
  - Signaler
  - Stakk
    → Automatisk utvidelse av stack

Synkronisering (interprosesskommunikasjon)

Forskjellige behov for synkronisering

1) **Hvordan kommunisere?**
   - Delt minne
   - Meldinger

2) **Ikke gå i bena på hverandre (race conditions):**

For å forhindre race conditions kan man bruke synkroniseringsmekanismer som Mutex, semafor og monitor

- **Mutex:** enten er en ressurs ledig eller ikke
- **Semaforer:** kan ha en mengde ressurser der vi kan si om det er noen ledige eller ikke. Typisk lesesaleksempel der vi har x leseplasser (ressurser) som en kø av studenter(prosesser) ønsker tilgang til. Derav Up() (en student opptar en leseplass) and Down() (En student forlater en leseplass).
- **Monitorer:** Er noe noen programmeringsspråk støtter. Feks det å låse hele objekter eller metoder.

3) **Sekvensiering**
   - Semafor
   - Conditionvariable

Kritiske regioner og delt minne

- **Kritisk område/region:**
  - En del av et program/tråd hvor delt minne aksseseres.
  - Tråder deler globale data i en prosess
  - Prosesser kan dele minnr ved å mappe inn delt minne i adresserommet.
- **Krav til god løsning av kritiske regioner**

  1) To prosesser kan ikke være samtidig i kritisk region
  2) Ingen antakelse om hardware kan gjøres om hastighet på og antall CPU'er
  3) Ingen prosesser som er utenfor kritisk region skal blokkere andre prosesser (bare den på insiden av huset som kan slippe inn andre).
  4) Ingen prosess må vente forgjeves for å komme inn i kritisk region

  → forgjeves → Utsulting
- **Løsninger:**
  - Gjensidig utelukkelse ved å skru av avbrudd (busy waiting)
  - Petersons solution
  - Gjensidig utelukkelse ved test-and-set-lock (TSL)
  - Semaforer
  - Monitorer + condition variable
  - Condition variable
  - Meldingssending
  - Javaspesifikke ting:
    - Synchronized
    - Objekt (Monitor)

- Runnable
- Wait() and Notify()
- Producer/Consumer

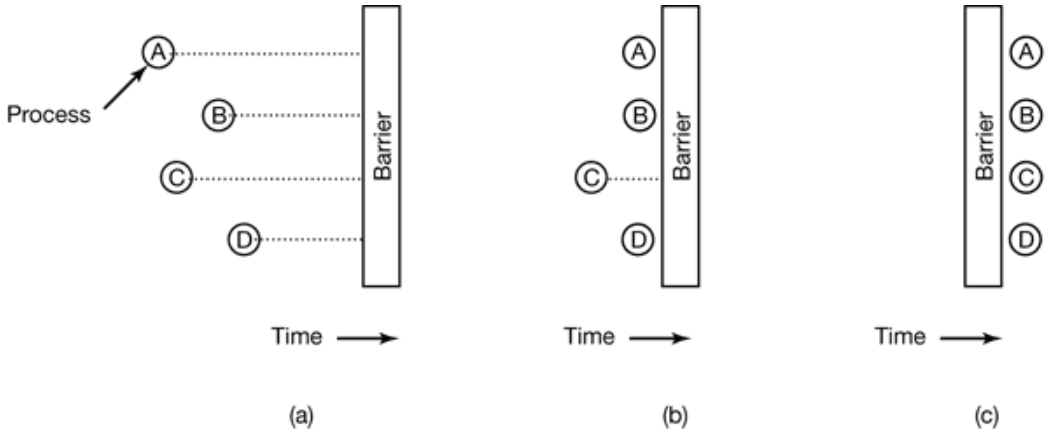
Meldingssending:

- Send(destination, message)
- receive(source, message
- systemkall i stedet for språkkostruksjon (som monitorer)
- Problemet med meldingssending er at en beskjed kan forsvinne på veien
  → Derfor sender mottaker ack tilbake.
  → Hvis ikke sender får ack innen et tildsinterval så sender den melding på ny.
- Hva om meldingen kommer frem, men ikke ack'en kommer frem?
  → Sender vil sende på nytt og mottaker vil få samme melding to ganger
- Problem kan komme med feks producer/consumer problemer der man må stoppe producer hvis den jobber fortere enn consumer og samme om consumer ikke ha noe å gjøre
  → Sett dem til å sove.

Guarded wait():

Barriere:

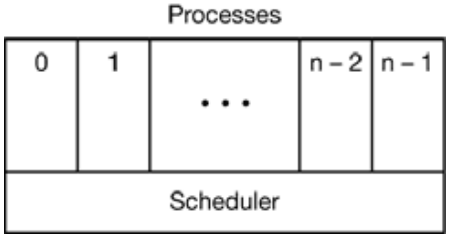
- Noen ganger må prosesser vente på hverandre. Dette kan løses med å sette opp en barriere.
- Når trenger vi barrierer?



Tidsdeling (scheduling):

Hvorfor tidsdeling?

- Det er flere tråder/prosesser i køen som ønsker å kjøre
- Tidsdeleren bestemmer hvem som er neste til å kjøre



Hva gjør tidsdeleren?

Når kjører tidsdeleren?

- Skaping av en prosess/tråd
- Terminering av prosess/tråd
- Blokkering av prosess/tråd
- Ved avbrudd (.. ved hvert n'te klokkeavbrudd)

(**Note:** Eneste måten å ta fra en prosess CPU er at det kommer et avbrudd)

Prosessoppførsel

- I/O bundet
- CPU bundet

Forskjellige typer tidsdelere

- **Innebygde tidsdelere:**
  Kjører som en del av prosessene og avbruddsrutiner
- **Separat prosess:**
  Aktiveres av timere

Avbruddbarhet (preemtion):

- **Avbrytbar** (preemtive) tidsdeling: bryt av ved hvert n'te klokkebrudd
- **Ikke avbrytbar** tidsdeling (non preemption): gir slipp på CPU ved blokering eller frivillig

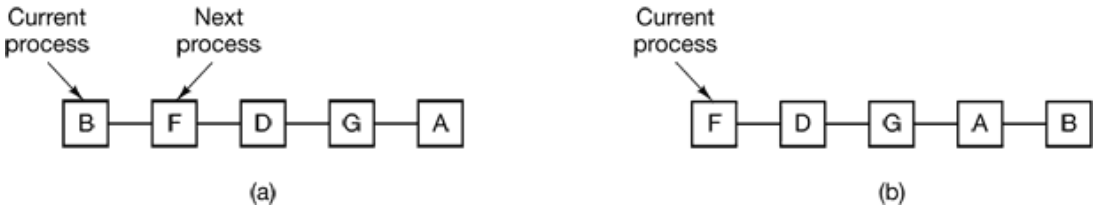
Scheduling algorithms:

- **FCFS - "First Come First Served" (FIFO)**
  - Non-preemtive
  - Blir tildelt CPU i den rekkefølgen de spurte etter det
  - Er en kø (FIFO)
  - Alle prosesser bruker CPU så lenge de trenger den (.. non-preemtion)
  - Sovene prosesser/tråder som våkner blir lagt bakerst i køen (gikk ut av kø--> må stille seg bakerst i køen igjen.. Ingen sniking her nei!)

    - (+) Lett å forstå og lett å programmere, “Rettferdig”
    - (-) Turnaround time (hva om en STOR prosess kommer først? → Mange prosesser må vente i køen lenge selv om de er små og tar liten tid)
- **SJF - “Shortest Job First**
    - Non-preemtive
    - (+) Turnaround time, responstid
    - (-) Krever at man har alle jobbene tilgjengelig (må vite run time)

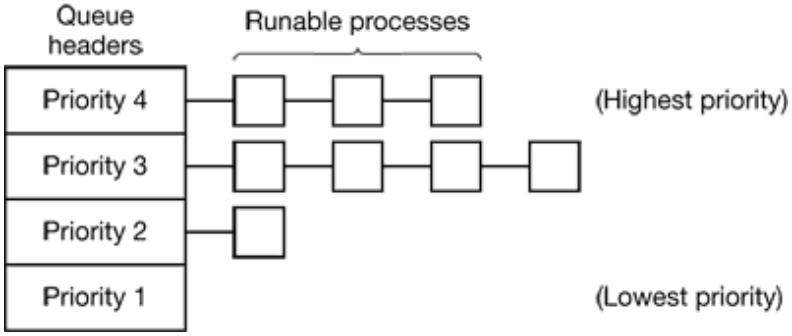
(a) feks FIFO, vanlig kjøring etter når de kom
(b) SJP, den korteste jobben kjører først.

- **RR - “Round Robin”**
    - Preemtive
    - Oldest, simplest, fairest and most widely used algorithm
    - Hver prosess får tildelt et tidsinterval (kvantum), hvor prosessen får kjøre.
    - Hvis prosessen fortsatt kjører i enden av intervallet kommer det et avbrudd og ny prosess kommer inn.
    - If the process has blocked or finished before the quantum has elapsed, the CPU switching is done when the process blocks, of course.
    - (-) Lengden på kvantum - kort, lang? Er vanskelig å si! Å bytte for mye frem og tilbake fører til mye tid brukt på å bytte prosess. Å sette for lang kvantum kan også føre til negative konsekvenser med tanke på at andre prosesser må vente lengre.
    - The conclusion can be formulated as follows: setting the quantum too short causes too many process switches and lowers the CPU efficiency, but setting it too long may cause poor response to short interactive requests.



- **Pri - “Prioritetskøer”**
    - Med RR antar man at alle prosesser er like viktige.
    - Hver prosess får tildelt en prioritet → de med høyest prioritet får kjøre først.
    - Hva om en prosess med høy prioritet kjører “uendelig” (lang tid)? Vil de med lav prioritet sulte (“starvation”)?
      → Løsning1: minke prioritet på prosess ved hvert klokketikk (klokke interrupt)
      → Løsning2: Alternatively, each process may be assigned a maximum time quantum that it is allowed to run. When this quantum is used up, the next highest priority process is given a chance to run.

    - Senk prioritet ved klokketikk, øk prioritet ved I/O
    - Prioritet kan bli gitt statisk eller dynamisk
    - (-) Kan føre til utsulting om de lavere prioritetene ikke blir behandlet!



(Her er prioritet med RR algoritme innenfor hver prioritet)

- **SRTN - "Shortest Remaining Time Next"**
    - Preemtive (the preemtive version of SJF!)
    - Det er den samme som SJF, bare at vi nå kan avbryte en kjørende prosess og bytte den ut med en som er kortere. Vi kan da få en rekkefølge som dette: A,B,C,B,D,B (B blir avbrutt siden kortere prosesser kommer inn)
- **Lotteri**
    - Hver tråd får lodd
    - Hver tråd kan oppnå flere lodd og får da mer CPU tid
    - Hver gang tidsdeleren kjører, trekkes et lodd
- **SPN - "Shortest process next"**

<u>Målet til tidsdelingsalgoritmer:</u>

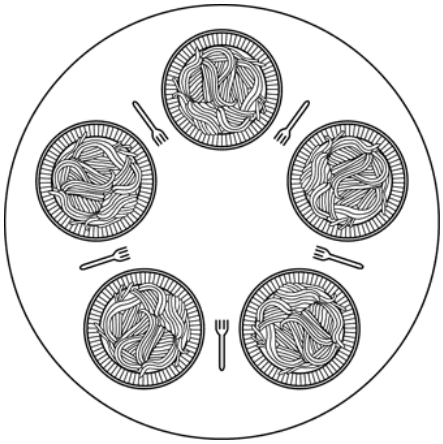
- All systems
    - Fairness - giving each process a fair share of the CPU
    - Policy enforcement - seeing that stated policy is carried out
    - Balance - keeping all parts of the system busy
- Batch systems
    - Throughput - maximize jobs per hour
    - Turnaround time - minimize time between submission and termination
    - CPU utilization - keep the CPU busy all the time
- Interactive systems
    - Response time - respond to requests quickly
    - Proportionality - meet users' expectations
- Real-time systems
    - Meeting deadlines - avoid losing data
    - Predictability - avoid quality degradation in multimedia systems

Tidsdelingsalgoritmer i forskjellige systemer:

- **Batch systems:**
  - FCFS/FIFO
  - SJF
  - SRTN
- **Interaktive systemer**
  - RR
  - Pri
  - Lotteri

Klassiske synkroniseringsproblemer:

- **Spisende filosofer**
  - Hver filosof kan tenke eller spise
  - Spising krever 2 gaffler
  - Kan bare ta opp en gaffel om gangen
  - Legger fra seg gafflene når man er ferdig å spise.



- **The readers and writers problem**
- **Sleeping barber problem**

Kapittel 3: “Memory Management”

3.1: Ingen minneabstraksjon **3.2: Minneabstraksjon: Adresseområde** **3.3: Virtuelt minne**	**3.4: Sideerstatnings-algoritmer** **3.5: Designvalg for sidedelt lager** **3.6: Impl.detaljer** **3.7: Segmentering**

Ingen minneabstraksjon:

Monoprogrammering

1. Bruker skriver kommando
2. OS’et kopierer programmet fra disk til minne
3. Kjører programmet
4. OS’et venter på neste kommando

Multiprogrammering

- Øker CPU-utnyttelsen
- Ha flere oppgaver om gangen
- Må håndtere relokering
  (Statisk og dynamisk relokering)
- When multiprogramming is used, the CPU utilization can be improved:

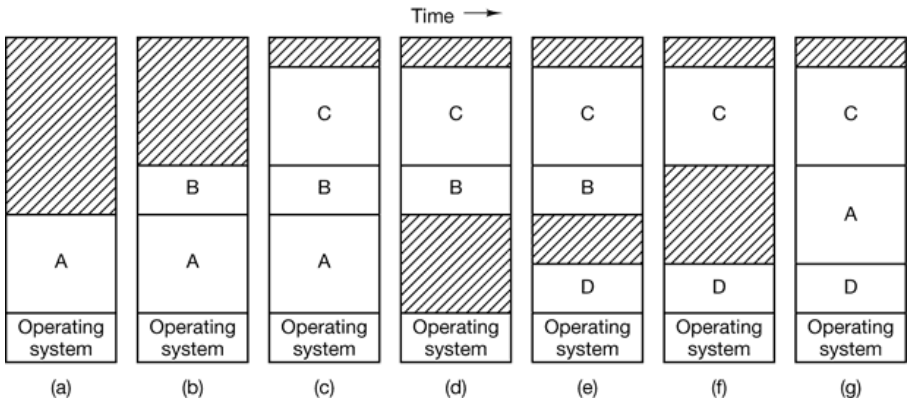
Minneabstraksjon: Adresseområde

Relokering av dynamisk binding av adresser:
- Base and limit registers!

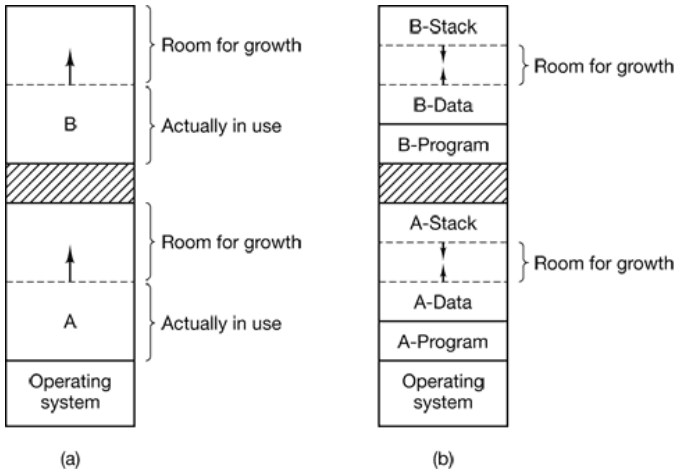
Kompilering, lenking og lasting

Swapping

- Two general approaches to dealing with memory overload: Swapping and virtual memory
- Enkel løsning for store minnebehov
- Swapping consists of bringing in each process in its entirety to main memory , running it for a while, then putting it back on the disk.



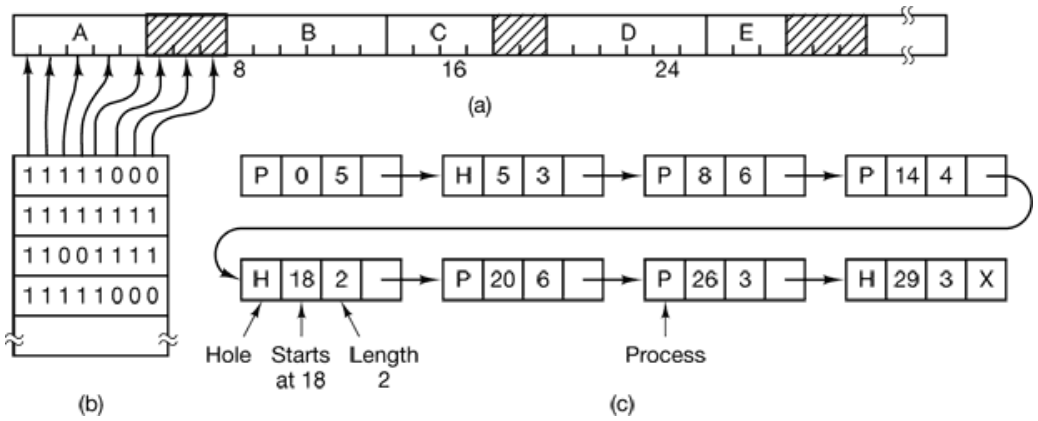
- **Growing need for space:** A point that is worth making concerns how much memory should be allocated for a process when it is created or swapped in. If processes are created with a fixed size that never changes, then the allocation is simple: the operating system allocates exactly what is needed, no more and no less. Det er kostbarn a swappe inn og ut prosesser på det grunnlag av at de trenger større plass.



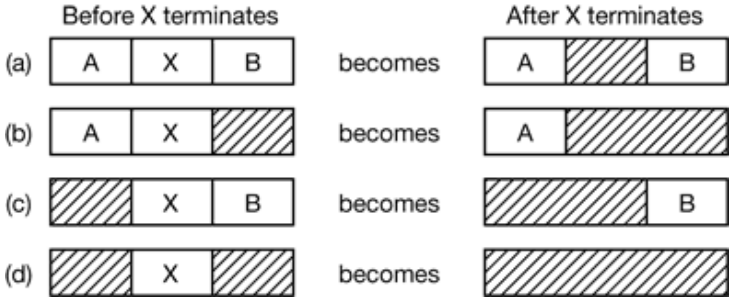
(a) Allocating space for growing data segment
(b) Allocating space for a growing stack and a growing data segment

Håndtering av leding minne: When memory is assigned dynamically, the operating system must manage it. In general terms, there are two ways to keep track of memory usage:

1. **Bitmaps**



2. **Lenkede lister**



Algoritmer for å finne ledig plass: When the processes and holes are kept on a list sorted by address, several algorithms can be used to allocate memory for a newly created process (or an existing process being swapped in from disk). We assume that the memory manager knows how much memory to allocate.

- **First fit:**
  - Simpel, tar det første og beste hullet som er stort nok
  - Når den finner et hull som er stort nok deler den hullet opp i to biter: minne den skal bruke og minne den ikke trenger (nytt hull).
- **Next fit:**
  - Samme som first fit, bare at den husker hvor den var sist og da tar neste beste plass den finner.
- **Best fit:**
  - Søker gjennom hele lista for å finne det minste passende hullet
  - Treg
  - Lager små ubrukelige hull
- **Worst fit:**
  - Tar alltid det største hullet slik at restene er store nok til at andre prosesser får bruk for det
    → Løser problemet til Best fit

    - Lager gode hull
- **Qick fit:**
    - Bruker separate lister for forskjellige klasser av størrelse på hull.
      Kan f.eks ha en liste med hull på størrelse 4KB, 8KB, etc.
    - Lett og raskt å finne et ledig og passende hull
    - Får samme problemet som best fit → står igjen med mange ubrukelige hull

Kommentarer til algoritmene:

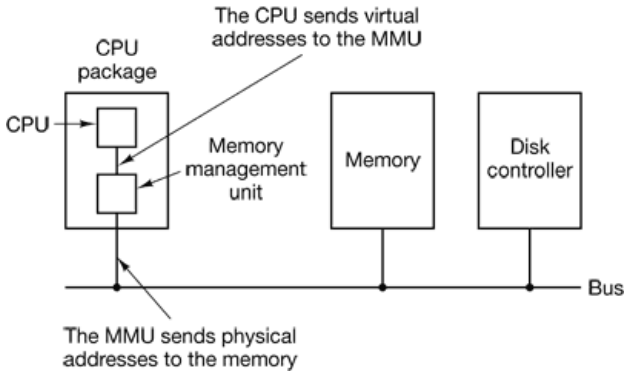
- Hvis ikke merging blir gjort på de små “restminnnene” så vil man bare sitte igjen med hull som er for små.
- Alle algoritmene kan effektiviseres med å ha separate lister for prosesser og hull.

Virtuelt minne

Hva er virtuelt minne?: The basic idea behind virtual memory is that the combined size of the program, data, and stack may exceed the amount of physical memory available for it. The operating system keeps those parts of the program currently in use in main memory, and the rest on the disk. For example, a 16-MB program can run on a 4-MB machine by carefully choosing which 4 MB to keep in memory at each instant, with pieces of the program being swapped between disk and memory as needed.

Hvorfor virtuelt minne?

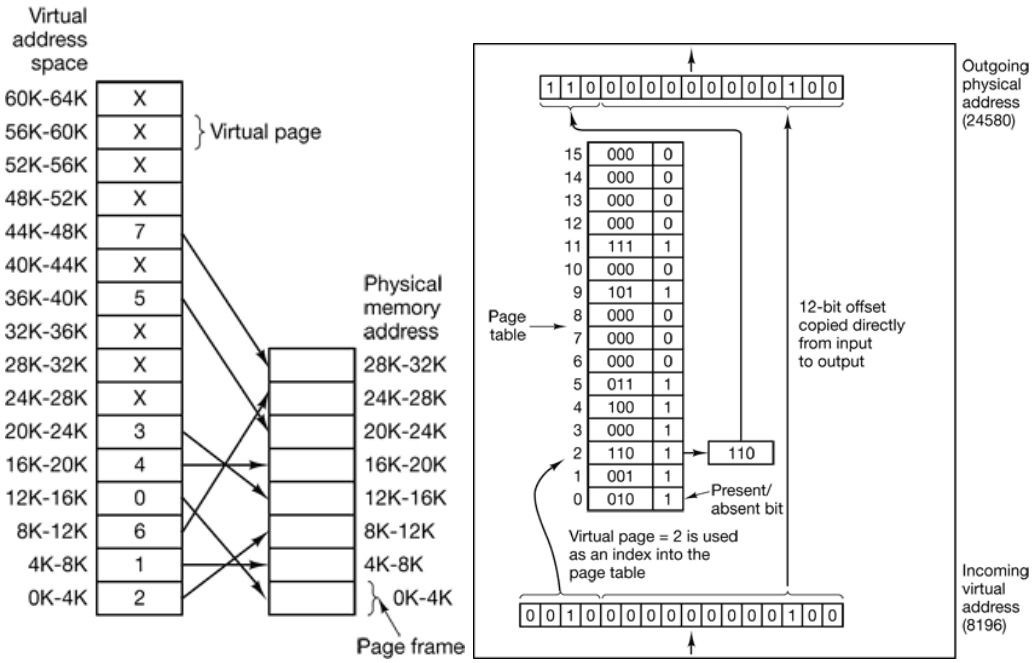
- Gjemmer fysisk minne
- Tillater deling av minne
- Tillater større adresserom enn fysisk minne
- Tillater større grad av multiprogrammering.
- Beskyttelse mellom prosesser



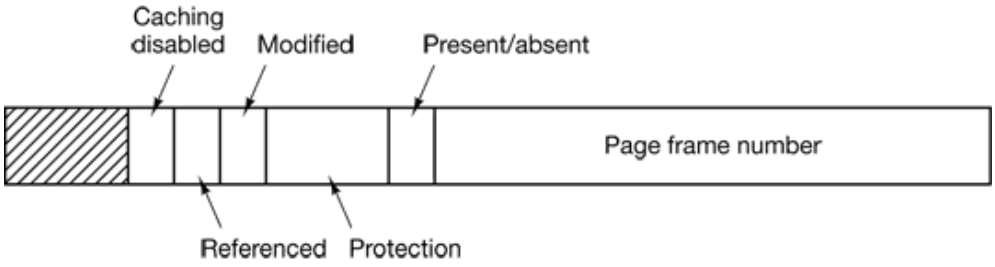
(The relation between logical (virtual) and physical memory)

Paging (sidedeling)

- The virtual address space is divided up into units called **pages**. The corresponding units in the physical memory are called **page frames**. The pages and page frames are always the same size
- **Present/absent bit:** Keeps track of which pages are physically present in memory
- **MMU** - “Memory Management Unit” (Til høyre)
- What if the MMU notice that a page is unmapped in memory?
  → **Trap (page fault)**



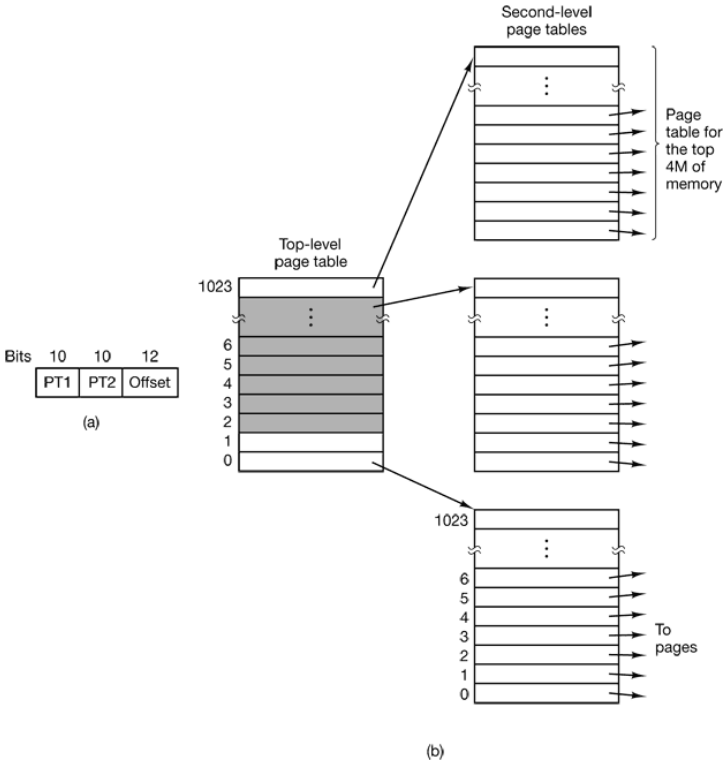
- **Structure of a page table entry:**



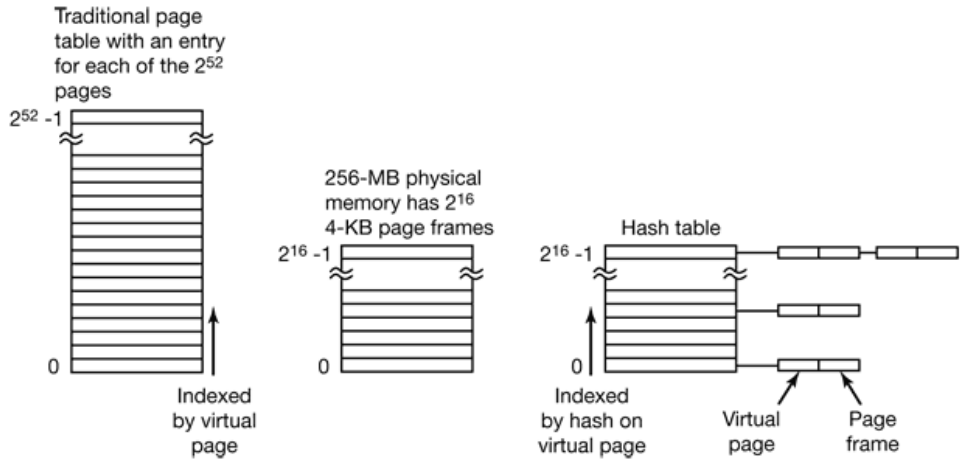
- **Datastrukturer for mapping mellom virtuelt og fysisk minne:**
  - **Sidetabell (Forward table)**
  - **Inverterte sidetabeller (Hashtable)**
  - **Lineære sidetabeller**
- **Problems we have to face with paging:**
  1. The mapping from virtual address to physical address must be fast.
  2. If the virtual address space is large, the page table will be large

- **Speeding up paging:**
  - **TLB** - “Translation Lookaside Buffer”
    - Software TLB - management: I stedet for at TLB er programmert inn på CPU så blir TLB behandlet av OS’et.
    → Mye enklere MMU
    → OS’et får et TLB-avbrudd og oppdaterer TLB’en
    - **Soft miss:** TLB-avbrudd. Oversettingen finnes ikke i minne. TLB kan oppdateres fra minne
    - **Hard miss:** Oversettingen finnes ikke i TLB, siden er ikke i minne og en sidefeil og en disklesning må til for å få siden inn i minne.

- **Page tables for large memories**
  - **Multilevel page table**
    - Store adresserom går enorme sidetabeller
      → Lag flere nivåer i tabellene.
    - Trenger kun deler av sidetabellen i minne samtidig
    - Nymoderne OS har 4 nivåer



- **Inverted page table:**
  - Hasher på virtuelle adresser, prosess id



(Comparison of a traditional page table with an inverted page table.)

Global vs Lokal sideerstatningsalgoritme:

- **Global:**
  - Fast antall rammer delt mellom prosesser
  - Alle prosesser kan stjele fra hverandre
  - Algoritmer: Optimal, NRU, LRU, NFU, FIFO, Secon Chance, Clock og Aging
- **Lokal:**
  - Hver prosess har et arbeidssett av sider
  - Stjeler fra seg selv
  - Algoritmer: Arbeidssett, WSClock (Workingset + Clock)

Sideerstatningsalgoritmer:

- **Optimal:**
  - "Samme som" FIFO, bare at den optimale ser frem i tid (prøver å unngå å bytte ut de sidene som skal brukes i fremtiden)
  - Brukes til sammenligning av algoritmer
  - (-) Kan være vanskelig å se frem i tid
  - (-) God i teorien, men kan ikke implementeres i praksis
- **NRU - "Not recently used":**
  - Bruker referenced bit (r) og modified bit (m) som settes av MMU og periodevis slettes av OSet.
  - 4 klasser:
    - 1) (ikke R, ikke M)
    - 2) (Ikke R, M)
    - 3) (R, ikke M)
    - 4) (R,M)
  - Velger en TILFELDIG side fra den laveste klassen og bytter ut
- **LRU - "Least recently used":** Bytter ut de sidene som har vert i lenge lengst eller som

ikke har blitt referert til.

- **NFU (Not Frequently Used):**
- **FIFO:**
    - Den siden som har vert lengst i minne byttes ut
    - (-) Den som har vert i minne lengst kan ofte være i bruk.
    - (+) Funker bra med random (uniform) aksess
- **Second Chance:**
    - FIFO hvor siden flyttes til slutten hvis R-bitene er satt ved sidefeil
    - R-biten nullstilles når siden flyttes bakerst
    - (-) Lister er “tungt”
- **Clock:**
    - Samme som second chance, men organiserer sidene som ei klokke og bruker en viser(teller) som startsted
      → Slipper listeoperasjoner (og fysiske pekere)
    - Kan ha klokke med to visere
- **Aging:**
- **Working set**
- **WS Clock (Working set + clock)**

Design issues for paging systems:

Lokal vs. global sideerstatning
Lastkontroll og thrashing
Optimal sidestørrelse
Fragmentering

- Intern fragmentering
- Ekstern fragmentering
- Data fragmentering
- Overhead

Deling av minne

Backingstore (Swap)

Implementationdetaljer

Segmentering

Hva er segmentering?

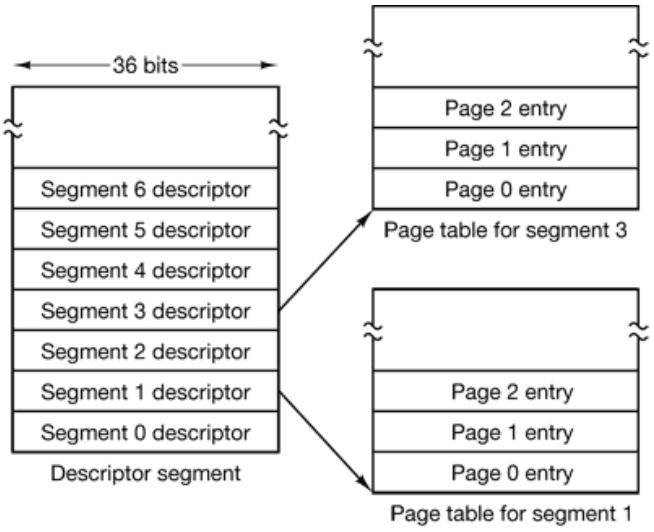
- Er et alternativ til paging
- I tillegg til minneadresse har vi et segmentregister/deskriptor

Hvorfor segmentering?:

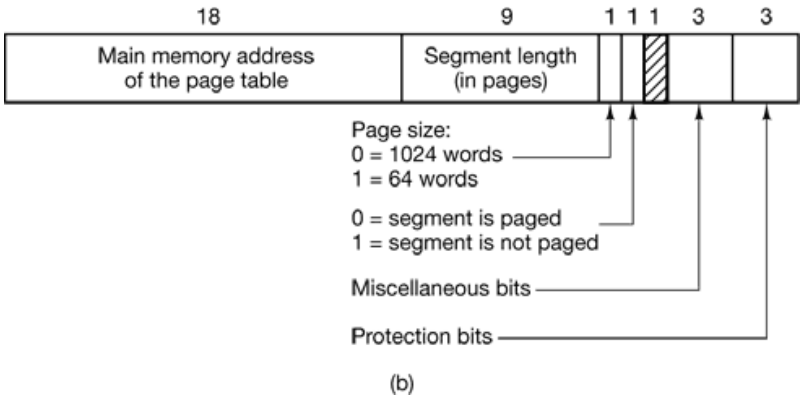
- Tillater deling av minne mellom prosesser
- Bedre mekanisme for beskyttelse
- (-) Mer komplisert for OSet og komplilatorene.

Segmentering vs. paging:

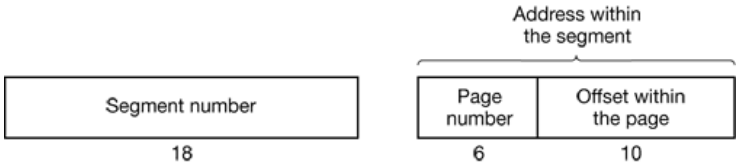
Segmentering med paging (multics):



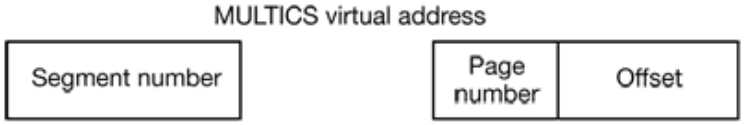
(a)

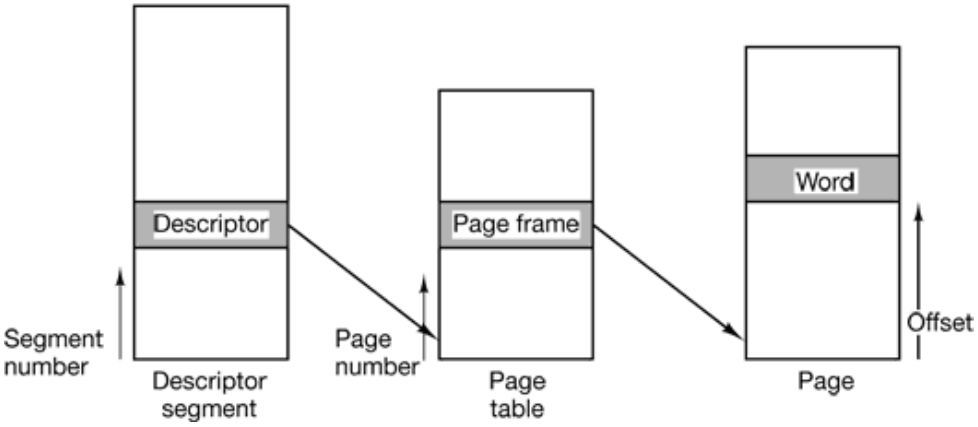


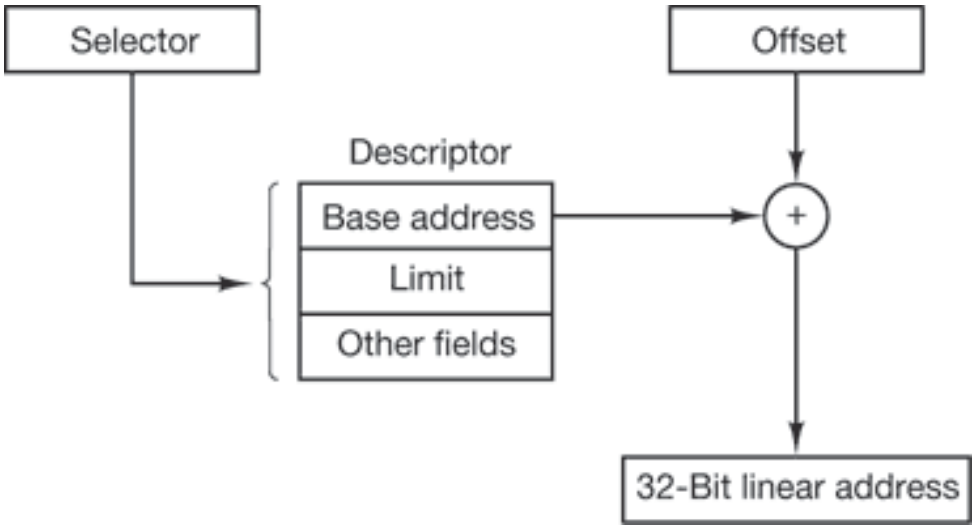
(b)



Virtual multics address







Kapittel4: “File Systems”

4.1: Files
4.2: Directories
4.3: File System implementation
4.4: File System Management and otimization

Motivation:

- Virtual address space limited during process execution
- RAM is volatile
- Often many processes need to access the same data
- Need for long-term information storage systems
    - Storing vert large amount of information
    - Information stored must survive after the termination
    - Accessing from multiple processes

Files:

File:

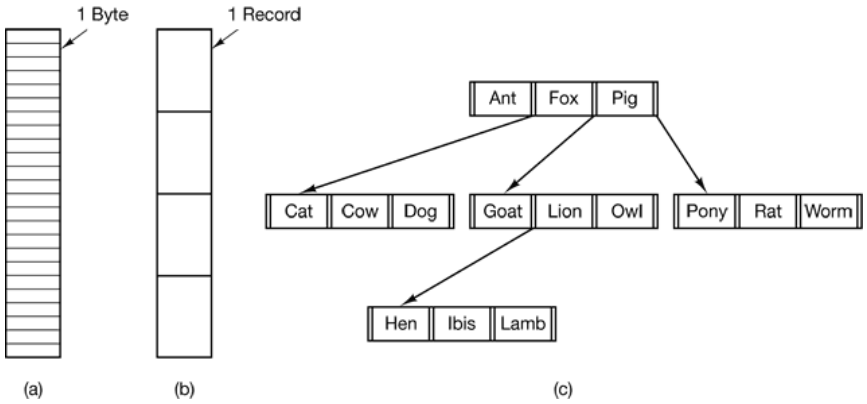
- Logical unit created by processes
- Perstistence
- Managed by OS
- File System: part of OS dealing with th files
    - Storing
    - Retrieving
    - Updating
    - Deleting
    - Managing space on disk

File Naming: Variablene under kan variere fra forskjellige system.

- Size of the name
- Characters permitted
- Case sensitivity
- Two-parts file name (name + extension).
  Here are some extensions:
    - .jpg, .gif → Bilder
    - .c, .py
    - .txt

File Structure: Files can be structured in any of several ways:

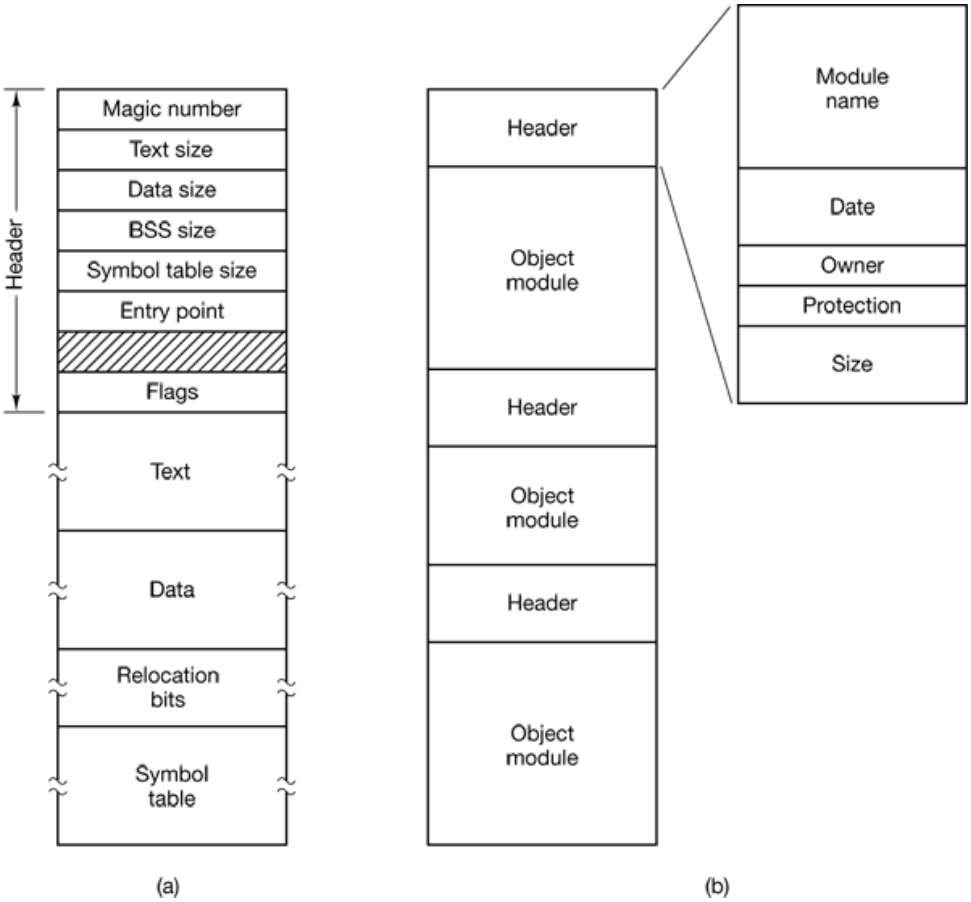
- **Byte sequence:** unstructured sequence of bytes. In effect, the operating system does not know or care what is in the file. All it sees are bytes. Any meaning must be imposed by user-level programs. Both UNIX and Windows use this approach.
- **Record sequence:** In this model, a file is a sequence of fixed-length records, each with some internal structure.
- **Tree:** In this organization, a file consists of a tree of records, not necessarily all the same length, each containing a key field in a fixed position in the record. The tree is sorted on the key field, to allow rapid searching for a particular key.



(a) Byte sequence. (b) Record sequence. (c) Tree.

File types:

- **Regular files** (ASCII or Binary): are the ones that contain user information.
- **Directories** (Structure of the system): are system files for maintaining the structure of the file system
- **Character special files** (I/O related): are related to input/output and used to model serial I/O devices such as terminals, printers, and networks.
- **Block special files** (Used to model disks)



(a) An executable file. (b) An archive.

File Access:

- **Sequential Access Files:** In these systems, a process could read all the bytes or records in a file in order, starting at the beginning, but could not skip around and read them out of order.
- **Random Access Files:** Files whose bytes or records can be read in any order
    - Seek operation

File attributes:

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/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary 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 has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

File attribute categories: Files protection, Flags, Only in file whose record can looked up by a key ,Various.

File Operations:

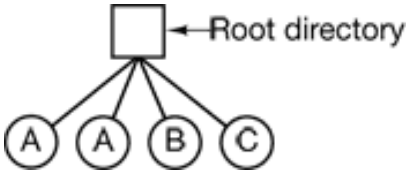
- Create → File created with no data
- Delete
- Open → To allow the system to fetch the attributes and list of disk addresses in main memory for rapid access in late calls
- Close → Force writing last block on physical memory
- Read → From current position the caller
- Write → Write file at current position
- Append → Add data to the end of the file
- Seek → For random access data only
- Get attributes
- Set attributes
- Rename

Directories:

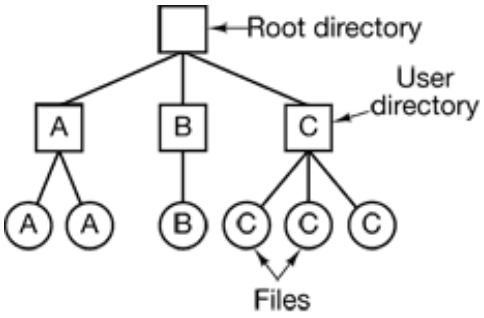
Directory: To keep track of files, file systems normally have directories or folders, which, in many systems, are themselves files.

Directory Systems:

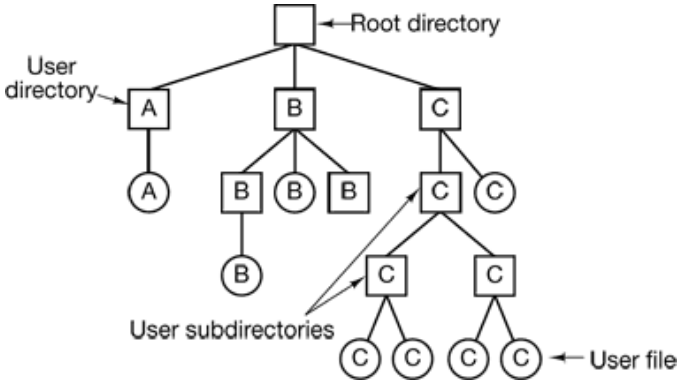
- **Single level:**



- **Two-level:**

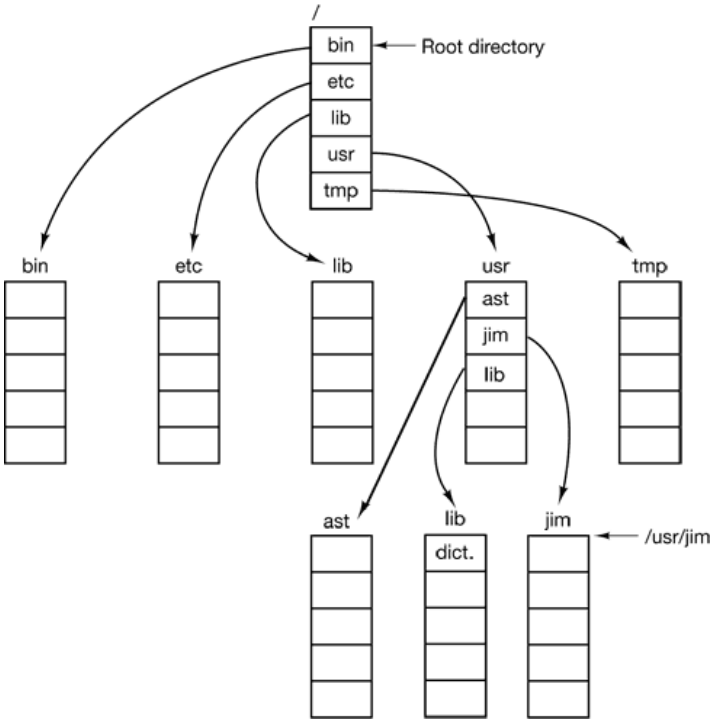


- **Hierarchical:**



Path Names:

- **Absolute:**
  - /usr/tmp/lab/mailbox
  - Path from the root
- **Relative**
  - Working directory
- **Special entries**
  - ".." - parent directory
  - "." - current directory

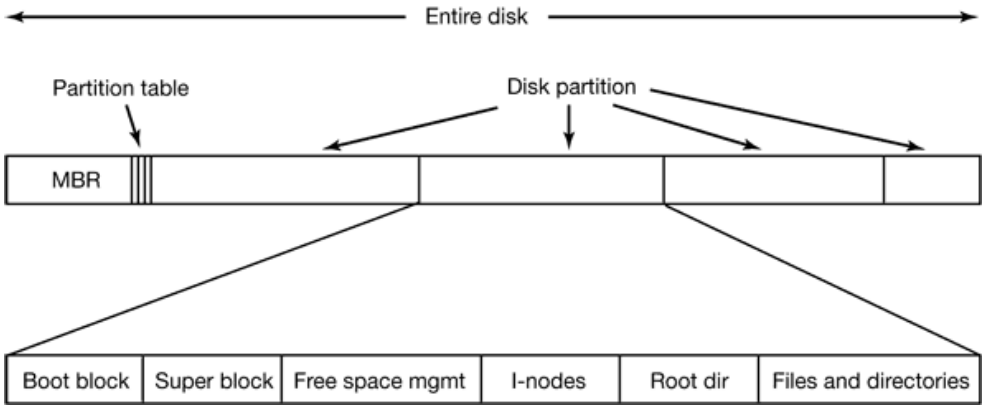


Directory Operations:

- Create
- Delete
- Opendir
- Closedir
- Readdir
- Rename
- Link
- Unlink

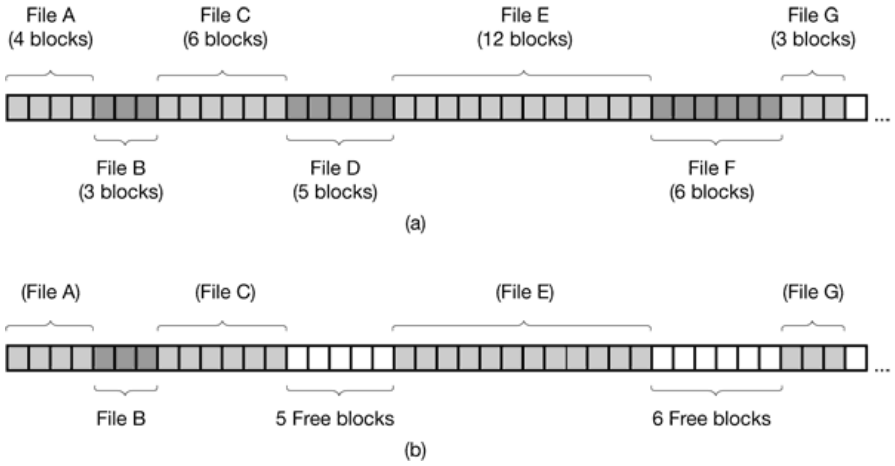
File System Implementation:

File System Layout:

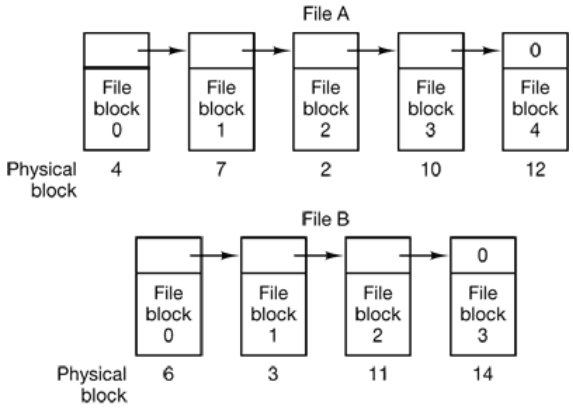


Implementing Files:

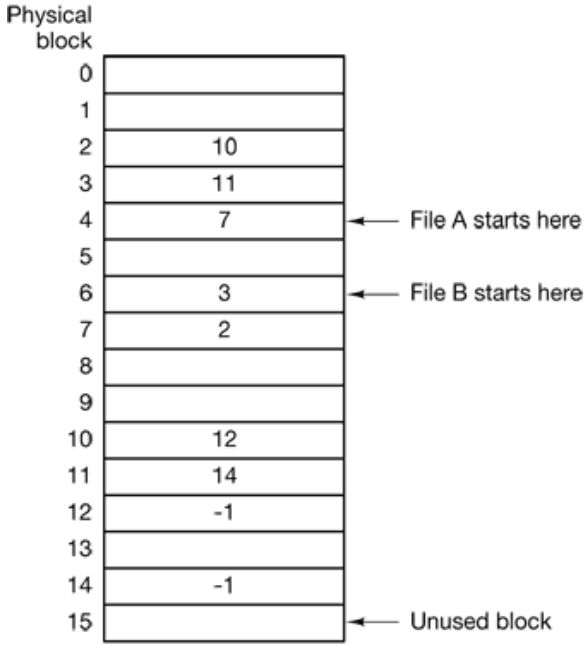
- **Contiguous Allocation:**
  (+) Simple implementation
  (+) Excellent read performance
  (-) Fragmentation (ubrukt plass)



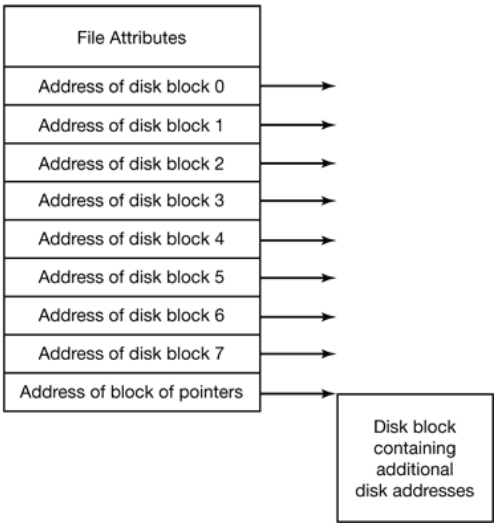
- **Linked List Allocation:**
  (+) No fragmentation
  (-)Slow random access
  (-)Extra overhead due to pointer space in block



- **Linked List Allocation using a table in memory(FAT):**
  (+) Hele blokka brukes til data
  (+) Easy random access
  (-) Hele tabellen in main memory
  (-) Kan ikke ha hull

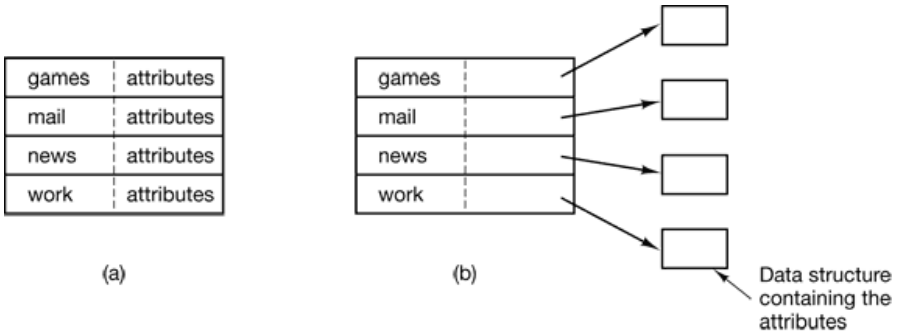


- **I-noder:**
  (+) Fixed size
  (+) In memory only when the file is open
  (-) If the disk addresses of a file exceeds over the fixed limit



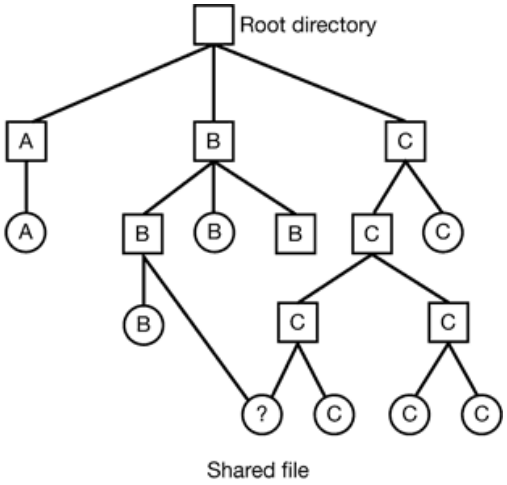
Implementing Directories:

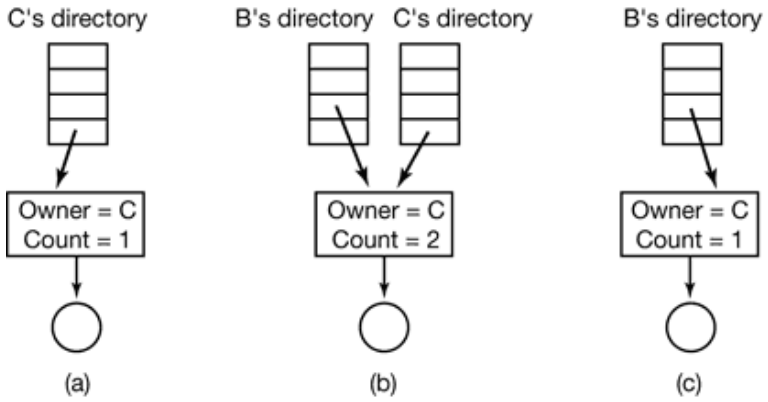
- Where to store file attributes
    - In directory entry
    - In the I-nodes



Shared Files:

- A file appear simultaneously in more than one directory (DAG)





(a) Situation prior to linking. (b) After the link is created. (c) After the original owner removes the file.

Journaling File Systems:

Virtual File Systems:

File System Management and Optimization:

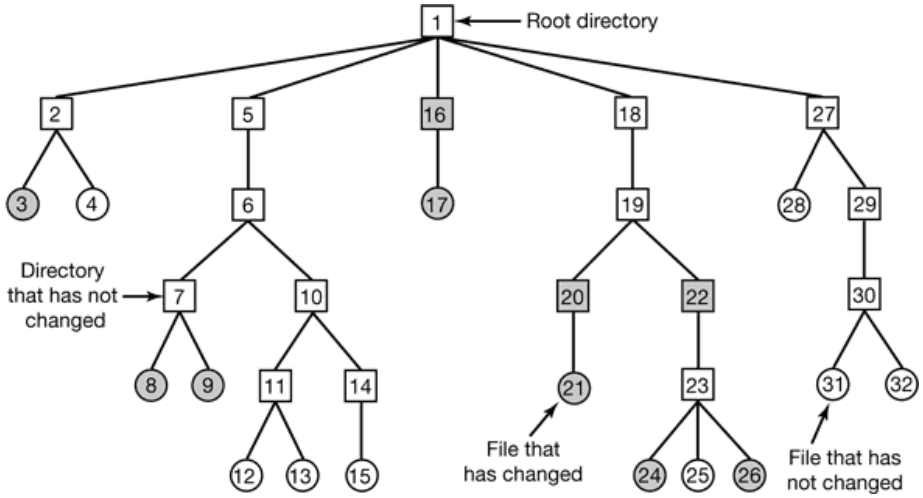
Disk space management:

- Large blocks: waste of space
- Small blocks: waste of time
- Performance and space utilization are in conflict
- Keeping track of free blocks
    - Linked link of blocks
    - Bitmap
- Disk quotas

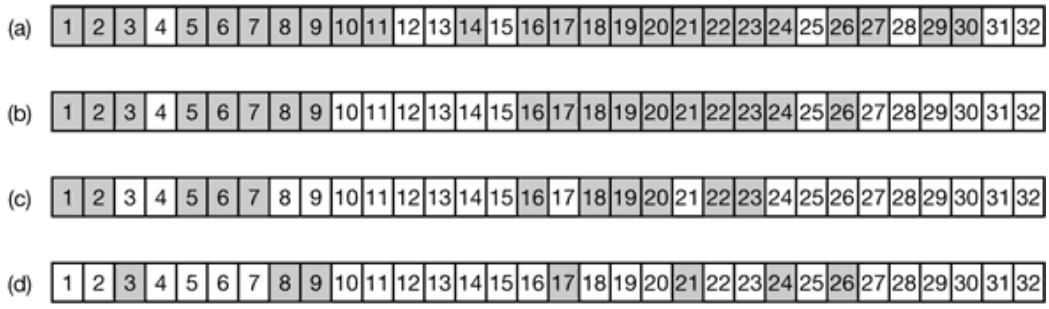
File System Backups:

- Destruction of a file system can be catastrophic
- Make backup is a smart solution
- **Backup is not trivial:**
    - Need of time and space
    - Need to do it efficiently
- **Issues:**
    - Backup of all file or just a part of it?
    - Use of an incremental dump
    - Compress the data before the backup
    - Backup of active file system
- **Dumping disk on tape (logical dump):** A logical dump starts at one or more specified

directories and recursively dumps all files and directories found there that have changed since some given base date



A file system to be dumped. The squares are directories and the circles are files. The shaded items have been modified since the last dump. Each directory and file is labeled by its i-node number.

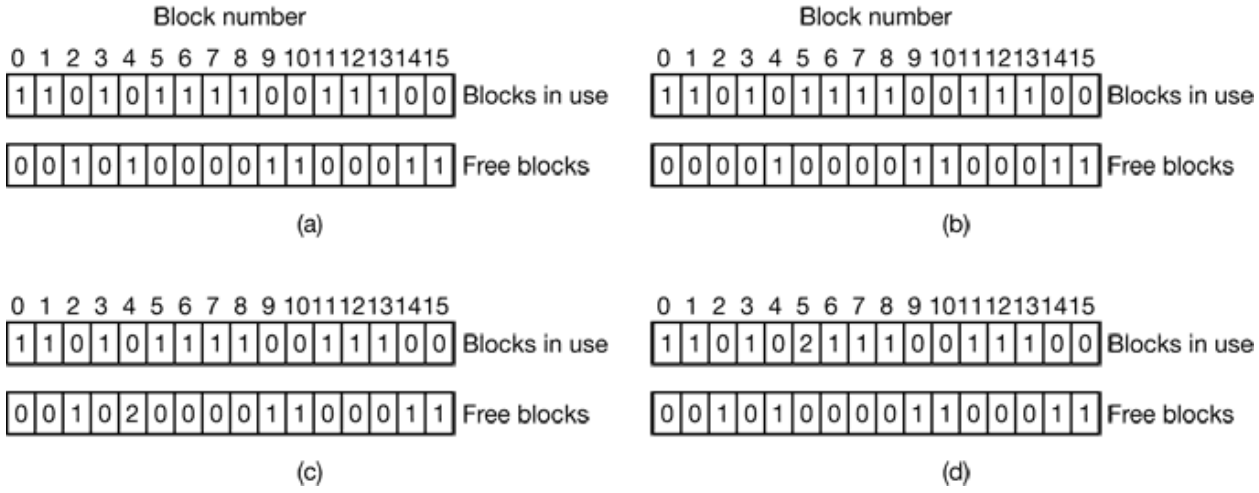


Bit maps used by the logical dumping algorithm.

1. **Phase 1** begins at the starting directory (the root in this example) and examines all the entries in it. For each modified file, its i-node is marked in the bitmap. Each directory is also marked (whether or not it has been modified) and then recursively inspected.
2. **Phase 2** conceptually recursively walks the tree again, unmarking any directories that have no modified files or directories in them or under them.
3. **Phase 3** consists of scanning the i-nodes in numerical order and dumping all the directories that are marked for dumping.
4. **Phase 4**, the files marked in Fig. 6-25(d) are also dumped, again prefixed by their attributes. This completes the dump.

File System Consistency:

- (a) Consistent
- (b) Missing block (block number 2)
- (c) Duplicate in the free list (block number 4)
- (d) Same datablock in more files (block number 5)



File system Performance:

- Caching
    - Block cache: collection of blocks that logically belong on the disk but are being kept in memory
    - Hash table quickly found if a block is present in the cache
    - Collision chains is kept to maintain the LRU order
    - Cache integrity in the face of a system crash.

Kapittel5: “I/O Management”

5.1: I/O maskinvare
5.2: I/O programvare
5.3: I/O programvarearkitektur
5.5: Klokker og timere

I/O Hardware:

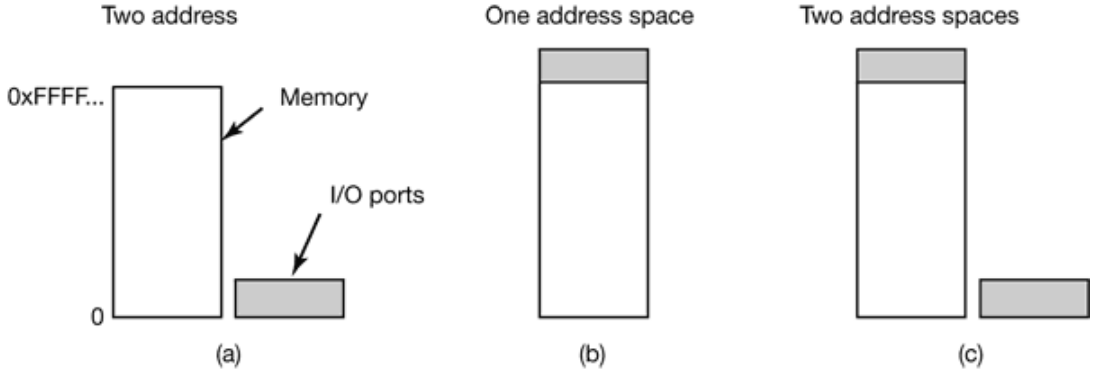
Klassifisering av I/O enheter

- Blokkerende: Disk, DVD
  (Kan lese/skrive blokker uavhengig av andre blokker → støtter seek)
- Tegnenheter: mus, tastatur (Ingen seek)
- Nettverksenheter

Kontrollere:

- Maskinvare som håndterer en eller flere mekaniske eller elektroniske enheter
- Konverterer en bit-strøm til/fra blokker/bytes
- Håndterer feilkorreksjon
- Kopierer til/fra primærlager
- Hver kontroller har noen få regsitre som er brukt for å kommunisere med CPU.
- Vi får et problem med hvordan CPU kommuniserer med kontrollregistrene og device data buffers. Her er to alternativer:
  - **Separat I/O:** Hvert kontrollregister får tildelt et I/O portnummer (fra et I/O port space) → Er beskyttet slik at bare OS kan få tilgang.
  - **Memory-mapped I/O:** Hvert kontrollregister er tildelt en unik minneadresse. Vanligvis er minneadressen på toppen adresseområdet.

Minneavbildet (memory-mapped) vs. separat I/O:



(a) Separate I/O and memory space. (b) Memory mapped I/O. (c) Hybrid.

Hvordan funker dette?:
How do these schemes work? In all cases, when the CPU wants to read a word, either from memory or from an I/O port, it puts the address it needs on the bus' address lines and then asserts a READ signal on a bus' control line. A second signal line is used to tell whether I/O space or memory space is needed. If it is memory space, the memory responds to the request. If it is I/O space, the I/O device responds to the request. If there is only memory space [as in Fig (b)], every memory module and every I/O device compares the address lines to the range of addresses that it services. It the address falls in its range, it responds to the request. Since no address is ever assigned to both memory and an I/O device, there is no ambiguity and no conflict.

Fordeler/Ulemper med minneavbildet I/O:

- (+) Kan bruke høynivåspråk for device drivers
- (+) Enklere beskyttelse
- (+) Mange instruksjoner

DMA

- Avbrudd og moderene CPU'er

I/O Software:

- Mål
- Programmert I/O med polling
- Avbruddsdreven I/O
- I/O med DMA

I/O software layers

- Interrupt handlers
- Device-drivers
- Deviceuavhengig I/O programvare

Klokker:

- Klokkehardware
- Klokkeprogramvare
- Soft timers

Kapittel6: “Deadlocks(Vranglås)”

Ressurser:

Ressurser: A resource can be a hardware device (e.g., a tape drive) or a piece of information (e.g., a locked record in a database). Vi har to klasser av ressurser:

- **Avbrytbare ressurser(preemtable):** A preemptable resource is one that can be taken away from the process owning it with no ill effects. Memory is an example of a preemptable resource.
- **Ikke avbrytbare(non-preemtable):** is one that cannot be taken away from its current owner without causing the computation to fail. If a process has begun to burn a CD-ROM, suddenly taking the CD recorder away from it and giving it to another process will result in a garbled CD

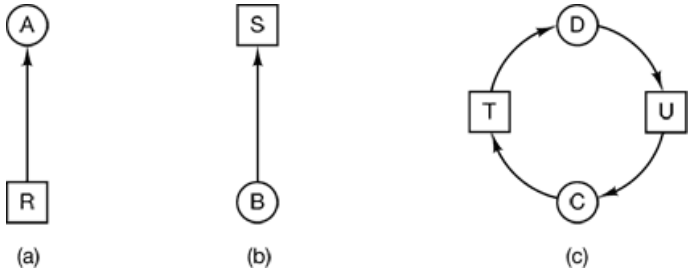
Hva gjør en ressurs om den ikke får ressursen den ber om?:

1. Kan sove til ressursen er ledig
2. Prosessen blir blokket
3. Prossessen venter og spørr om ressursen om en stund (er så godt som blokket siden den ikke får gjort noe nyttig mens den venter)

Vranglås:

Hva er en vranglås?

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.



(a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

Betingelser for vranglås: All four of these conditions must be present for a deadlock to occur!

1. **Mutual exclusion condition.** Each resource is either currently assigned to exactly one process or is available.
2. **Hold and wait condition.** Processes currently holding resources granted earlier can request new resources.
3. **No preemption condition.** 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 chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

Strategies for dealing with deadlocks:

1. Just ignore the problem altogether. Maybe if you ignore it, it will ignore you.
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 conditions necessary to cause a deadlock.

Strutsealgoritmen:

Algoritmen: Lat som om det ikke er noe problem! Den er faktisk vanlig i bruk.

Når brukes en slik metode?:

- Vranglås skjer sjeldent
- Mer kostbart å oppdage enn å bare ignorere

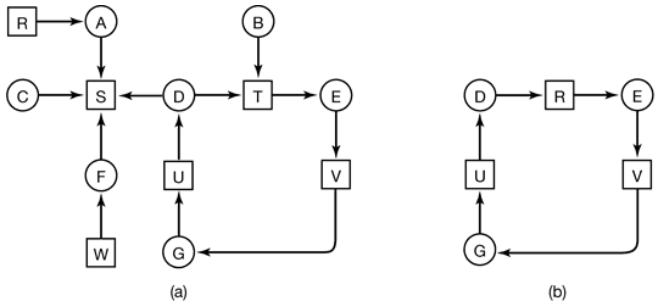
Når brukes ikke en slik metode?:

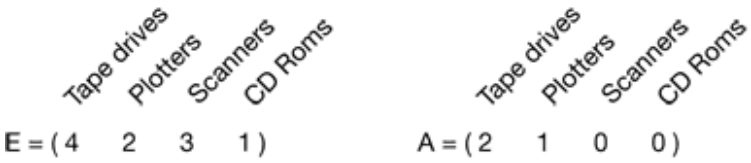
- Når vranglås oppstår ofte
- Når kritiske konsekvenser av systemkrasj kan forekomme (tap av sensitiv data feks)

Oppdagelse og recovery:

Vranglåsoppdagelse:

- … med bare **en ressurs** av hver type tilgjengelig (ressursgaf).
  Hvis man finner en sykel i ressursgrafen så har man en vranglås!
- … med **multiple ressurser** av hver type(Algoritme).
  - E = Totalressursvektor
    A = Tilgjengelige ressurser
    C = Gjeldende allokering (current matrix)
    R = Ønsket allokering (Requested matrix)
  - Alle prosesser som ikke “markeres”/fullføre er i vranglås.





Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Recovery ved vranglås:

- **Preemption:** Ta en ressurs fra en prosess å gi den til noen andre. Fortsette slik til (kanskje) vranglåsen løser seg (Ikke alltid mulig å stjele ressurser!)
- **Tilbakerulling:** If the system designers and machine operators know that deadlocks are likely, they can arrange to have processes checkpointed periodically. Kan rulle prosessen tilbake i en tilstand der det ikke var vranglås!(blir ofte brukt i kombinasjon med timeout → hvis en prosess ikke fullfører innen tiden så antar vi vranglås → tilbakerull)
- **Dreping av prosess:** En enkel (ond) måte å løse en vranglås på er å drepe en prosess i håp om at det skal løse opp vranglåsen. Man kan fortsette å drepe prosesser til vranglåsen løser seg.

Vranglåsunngåelse:

Safe/Unsafe state

- Mange av algoritmene for vranglåsunngåelse er basert på konseptet med safe/unsafe state.
- **Safe:** Har alltid nok ressurser slik at alle prosessene får sitt ønske.

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3
(a)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1
(b)

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5
(c)

	Has	Max
A	3	9
B	0	–
C	7	7

Free: 0
(d)

	Has	Max
A	3	9
B	0	–
C	0	–

Free: 7
(e)

- **Unsafe:** Ender opp i en tilstand der vi ikke har nok ressurser til å fullføre prosessers ønske om ressurser

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3
(a)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2
(b)

	Has	Max
A	4	9
B	4	4
C	2	7

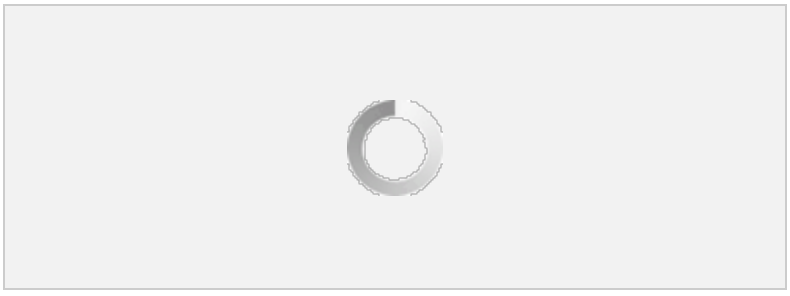
Free: 0
(c)

	Has	Max
A	4	9
B	–	–
C	2	7

Free: 4
(d)

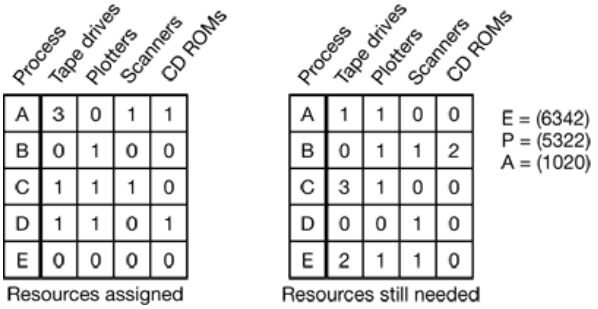
- **Banker's algorithm:**
  - Ulempe: Må vite alle ressursene på forhånd
  - Prinsippet er å holde seg i sikker tilstand hele tiden → Ikke ta valg som fører til usikker tilstand.

- **Banker's algorithm with one resource:**



(a) Safe. (b) Safe (c) Unsafe.

- **Bankers algorithm with multiple resources:**



Process	Tape drives	Plotters	Scanners	CD ROMs
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

Process	Tape drives	Plotters	Scanners	CD ROMs
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources still needed

E = (6342)
P = (5322)
A = (1020)

Vranglåsforhindring:

Vranglåsforhindring: Having seen that deadlock avoidance is essentially impossible, because it requires information about future requests, which is not known, how do real systems avoid deadlock? The answer is to go back to the four conditions to see if they can provide a clue. If we can ensure that at least one of these conditions is never satisfied, then deadlocks will be structurally impossible. ANGRIP de 4 betingelsene for vranglås:

- **Gjensidig utelukkelse:**
  - Bruk spooling
- **Hold og vent:**
  - Spør om alle ressursene er tilgjengelig før du starter.
  - (-) Ulempen er at det blir lite paralellitet.
  - (-) Alle prosesser vet ikke hvor mange ressurser de trenger
- **Ingen preemtion:**
  - Ikke mulig for alle ressurser
- **Sirkulær rekkefølge:**
  - One way is simply to have a rule saying that a process is entitled only to a single

resource at any moment. If it needs a second one, it must release the first one. For a process that needs to copy a huge file from a tape to a printer, this restriction is unacceptable.

- Alle spør om ressurser i samme rekkefølge → Ingen sykler

Relaterte tema:

Two-phase locking: A approach often used is called two-phase locking. In the first phase the process tries to lock all the records it needs, one at a time. If it succeeds, it begins the second phase, performing its updates and releasing the locks. No real work is done in the first phase.

Nonresource deadlock: All of our work so far has concentrated on resource deadlocks. One process wants something that another process has and must wait until the first one gives it up. Deadlocks can also occur in other situations, however, including those not involving resources at all. For example, it can happen that two processes deadlock each waiting for the other one to do something. This often happens with semaphores. In Chap. 2 we saw examples in which a process had to do a down on two semaphores, typically *mutex* and another one. If these are done in the wrong order, deadlock can result.

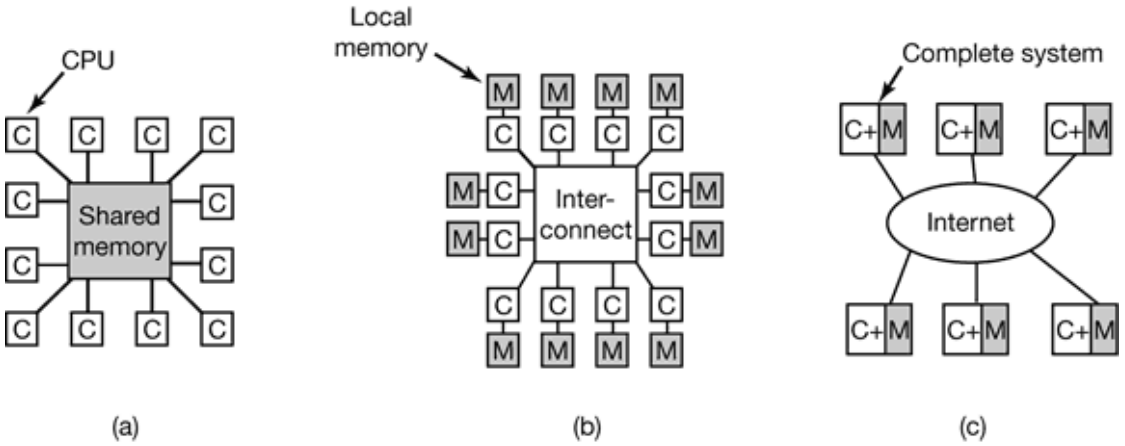
Starvation: Hvis en prosess aldri får den ressursen den trenger, men andre slipper forbi i "køen". Feks med shortest job first så kan lange prosesser bli sultet ihel hvid det konstant vil komme kortere prosesser som sniker forann dem i køen.
Vente i en evighet → Prosesser sulter ihel.

Kapittel8: “Multiprosessorsystemer”

8.1: Multiprosessorer
8.2: Multidatamaskiner/cluster
8.3: Virtualisering

Typer av multiprosessorer:

- (a) Multiple CPU’er med delt minne
- (b) Multiple CPU’er med egne minner



Multiprosessorer:

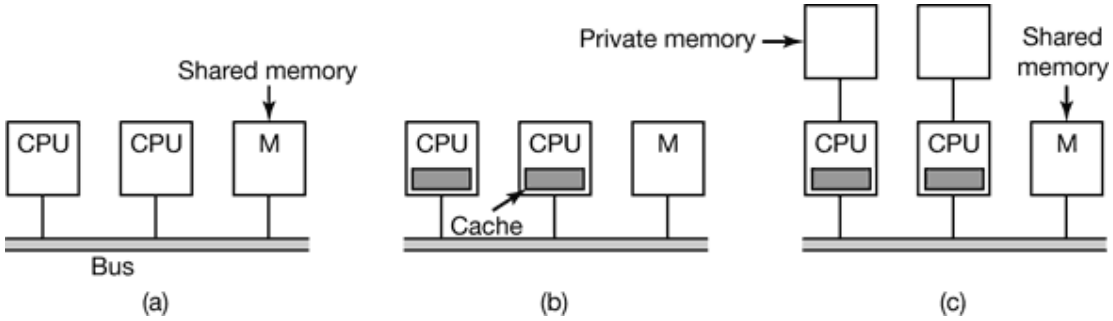
Multiprosessor: A shared-memory multiprosessor (or just multiprosessor) is a computer system in wich two or more CPUs share full access to a common RAM.

Multiprosessor hardware:

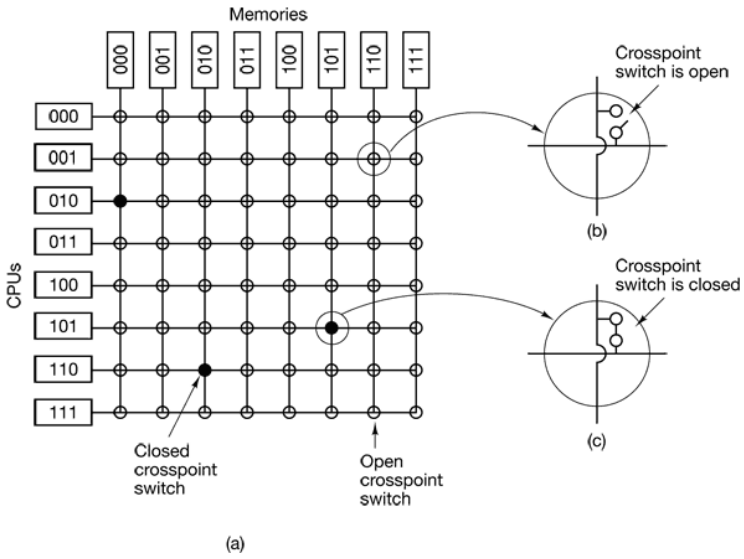
- **UMA** (Uniform memory access)
- **NUMA** (Nonuniform memory access)

UMA:

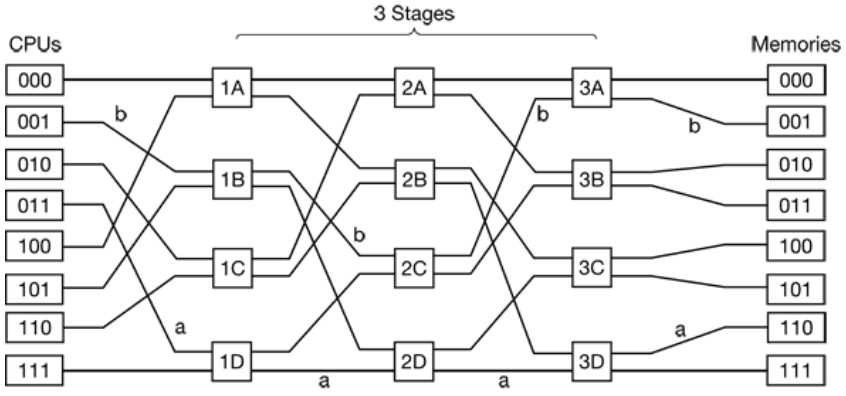
- **UMA med bussarkitektur**



- **UMA med crossbar switch (mesh interconnect)**

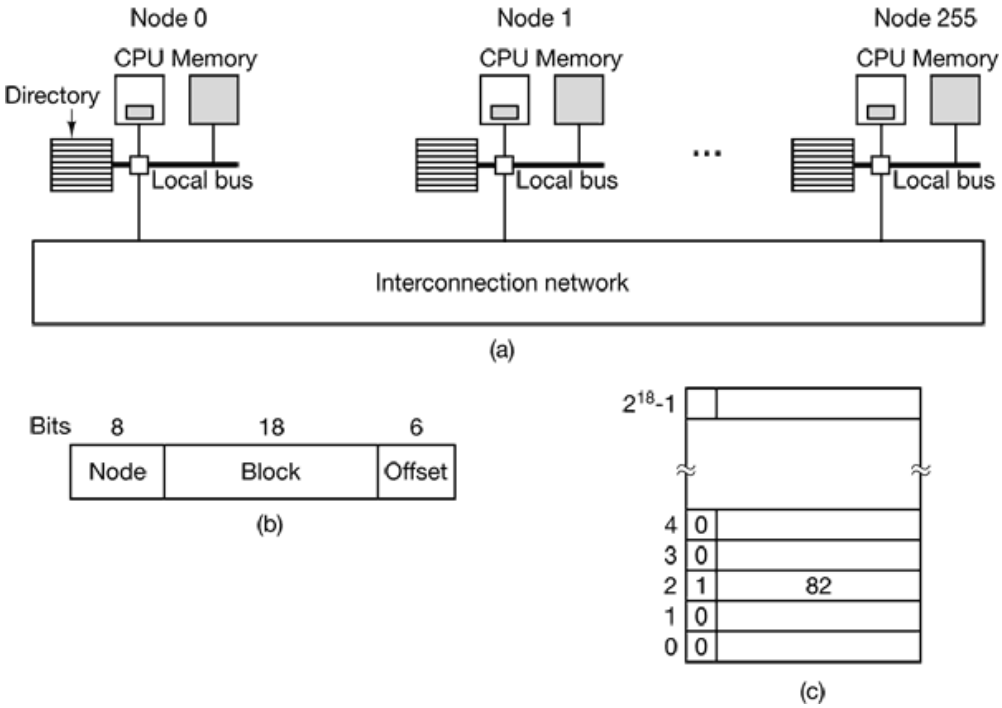


- **UMA med mangestegs switchet nettverk**



NUMA:

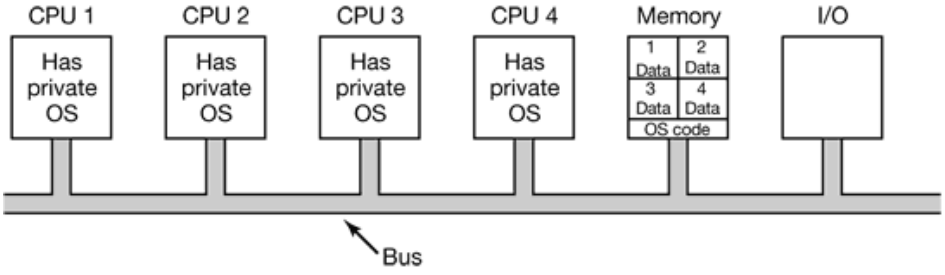
- Alle prosessene deler adresseområde
- Rask aksess til eget minne
- Treg aksess til fjernt minne
- NC-NUMA: ingen cache
- CC-NUMA: Numa med cache → Prøver å gjemme treg minne aksess



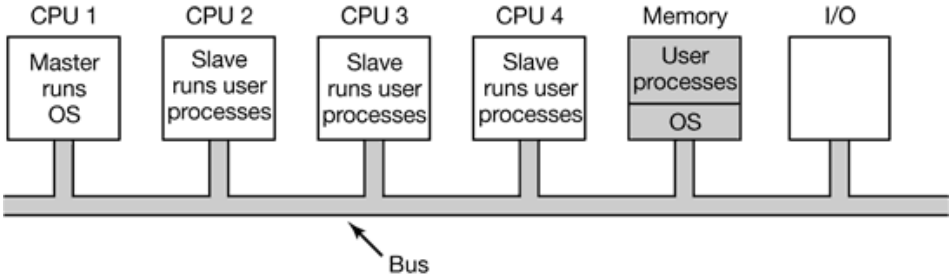
Multicore/Manycore:

Multiprosessor-OS-Typer:

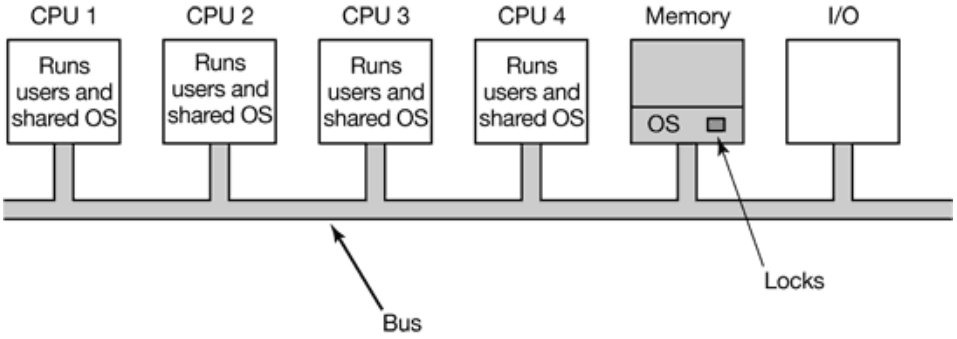
- **Hver CPU har eget OS**
  - Kan dele kode, men hver CPU har sitt eget dataområde
  - Systemkall håndteres av den CPUen som gjorde kallet
  - Hver CPU har sine egne OS-prosesser
  - Fast minneallokering
  - (-) Ingen deling av buffer cache, derfor brukes ikke denne modellen.



- **Master/slave multiprosessor**
  - Alle CPUer deler OS
  - (-) Ikke skalerbart fordi master blir en flaskehals

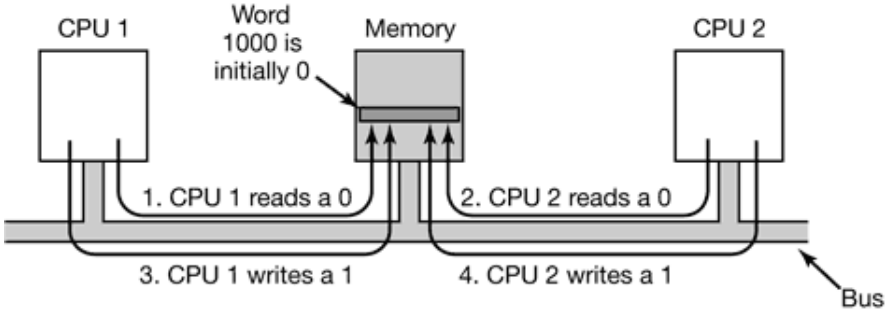


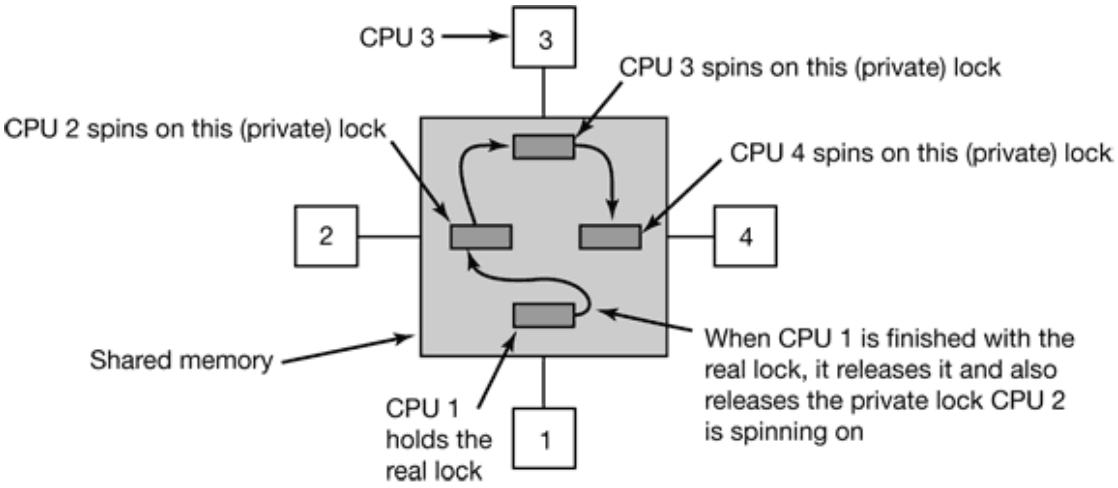
- **Symmetric multiprocessor (SMP)**
  - Ett OS i minne
  - Alle CPUene kan kjøre OSet
  - Må porsjonere OSet slik at flere CPUer kan kjøre OSet samtidig
  - Må synkronisere OSets datastrukturer
  - Kan få vranglåser



Synkronisering for multiprosessorer:

- Hvordan lage mutexer for multi-CPUer?
- Vi kan ikke bruke TSL





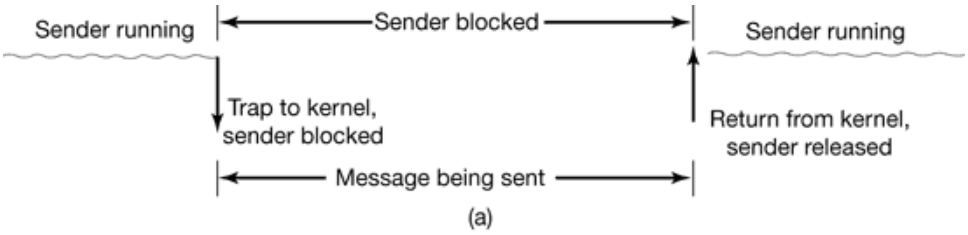
- **Spinning vs. switching:**
  - Alltid spinne (Ok når de spinner i kort tid)
  - Alltid svitsje (Svitsje til en annen tråd mens den andre spinner)
  - Observer spinnetid og bestemmer dynamisk.

Tidsdeling for multiprosessorer: Må ha kjernetråder for at OS skal kunne tidsdele de. Tidsdeleren må bestemme hvilken tråd som skal kjøre og hvilken CPU som skal kjøre trådene (time + space)

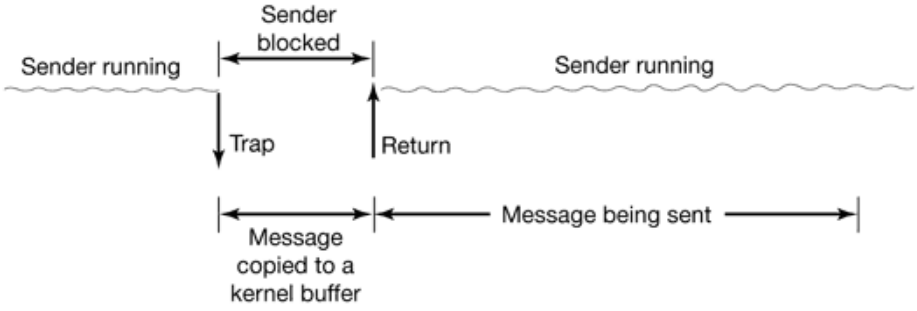
- **Timesharing (Time):**
  - Tidsdeling med delt kø:
  - Smart scheduling: Tråder kan flagge at de har spinlock
  - Affinity scheduling: La samme CPU kjøre de samme trådene.
- **Space sharing (Space):**
  - La en gruppe relaterte tråder kjøre samtidig
  - Venter å kjøre til man har nok CPUer tilgjengelig
  - Kjøre ferdig uten multiprogrammering
- **Gang scheduling (Space + Time):**
  - Relaterte tråd er kjører samtidig
  - Kjører i felles timeslice
  - Nyttig når trådene samarbeider mye

Multicomputes:

- Lavnivå kommunikasjonsprogramvare
  - Blocking vs. nonblocking call



(a)



(b)

  - The calls described above are **blocking calls** (sometimes called **synchronous calls**). When a process calls *send*, it specifies a destination and a buffer to send to that destination. While the message is being sent, the sending process is blocked (i.e., suspended). The instruction following the call to *send* is not executed until the message has been completely sent, as shown in Fig. 8-20(a). Similarly, a call to *receive* does not return control until a message has actually

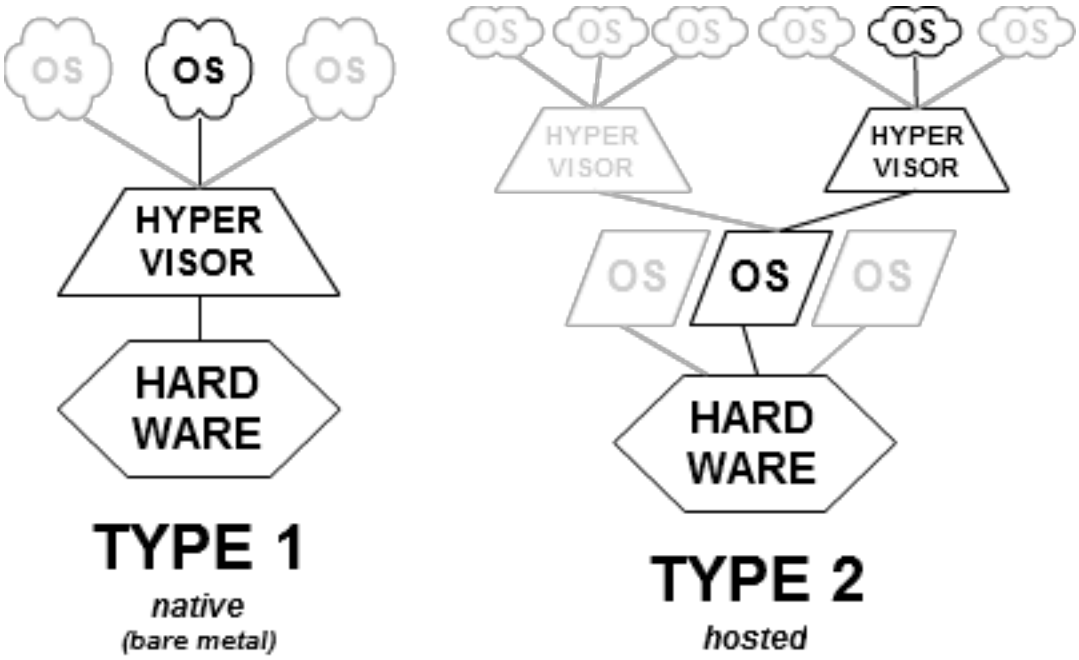
been received and put in the message buffer pointed to by the parameter. The process remains suspended in *receive* until a message arrives, even if it takes hours. In some systems, the receiver can specify from whom it wishes to receive, in which case it remains blocked until a message from that sender arrives.

    - An alternative to blocking calls are **nonblocking calls** (sometimes called **asynchronous calls**). If *send* is nonblocking, it returns control to the caller immediately, before the message is sent. The advantage of this scheme is that the sending process can continue computing in parallel with the message transmission, instead of having the CPU go idle (assuming no other process is runnable). The choice between blocking and nonblocking primitives is normally made by the system designers (i.e., either one primitive is available or the other), although in a few systems both are available and users can choose their favorite.
- Brukernivåkommunikasjon
- RPC - “Remote Precedure Call”
- Distibuert delt minne (DSM)
- Lastbalansering av prosesser

Virtualisering

A hypervisor, also called a virtual machine manager, is a program that allows multiple operating systems to share a single hardware host. Each operating system appears to have the host's processor, memory, and other resources all to itself. However, the hypervisor is actually controlling the host processor and resources, allocating what is needed to each operating system in turn and making sure that the guest operating systems (called virtual machines) cannot disrupt each other.

- Hypervisor type1
- Hypervisor type2
- Paravirtualisering



Kapittel9: “Security”

The Security Environment:

Threats:

- **Data confidentiality:** having secret data remain secret.
- **Data Integrity:** unauthorized users should not be able to modify any data without the owner’s permission.
- **System availability:** nobody can disturb the system to make it unusable.

Intruders: Selvfølgelig de vi skal beskytte oss mot. Inntrengere har ofte forskjellig mål og bruker forskjellige metoder. Noen er en trussel, mens andre bare vil ha det litt “gøy”.

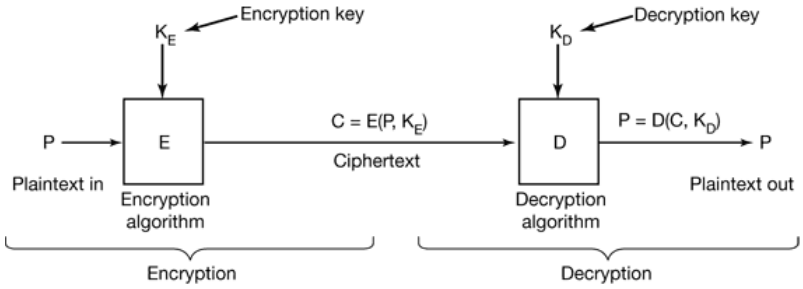
Accidential Data Loss:

- Acts of God (jordskjelv, flom, brann, etc.)
- Hardware or software error
- Human error

Cryptography

The basic: The purpose of cryptography is to take a message or file, called the plaintext, and encrypt it into the ciphertext in such a way that only authorized people know how to convert it back to the plaintext.

- Plaintext
- Chipertext
- Key
- **Encription: I**f *P* is the plaintext file, *KE* is the encryption key, *C* is the ciphertext, and *E* is the encryption algorithm (i.e., function), then *C* = *E*(*P*, *KE*). This is the definition of encryption.
- **Decryption:** *P* = *D*(*C*, *KD*) where *D* is the decryption algorithm and *KD* is the decryption key.



Secret-Key Cryptography(symmetric-key):

- **monoalphabetic substitution:**
  Plaintext: ABCDEFGHIJKLMNOPQRSTUVWXYZ
  Ciphertext: QWERTYUIOPASDFGHJKLZXCVBNM
- Her bruker man **en** nøkkel til å decrypt og encrypt.
- Det kan være lett å finne nøkler med feks bokstaver fordi noen bokstaver forekommer oftere enn andre. Kan da lett skrive algoritmer som "oversetter ".
- Sender og mottaker må sitte på en hemmelig nøkkel

Public-Key Cryptography:

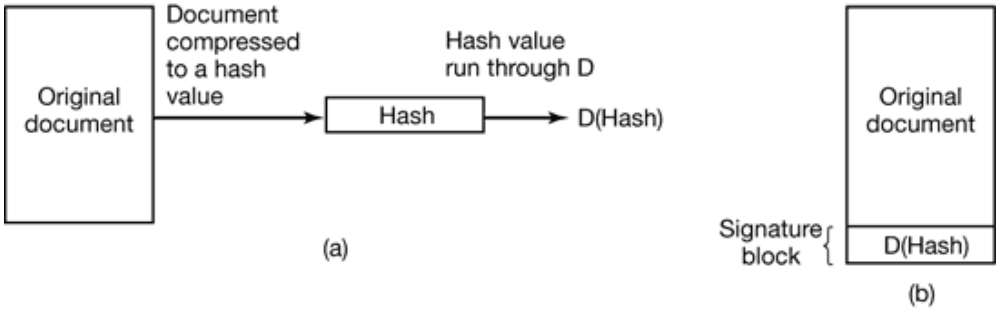
- This system has the property that distinct keys are used for encryption and decryption and that given a well-chosen encryption key, it is virtually impossible to discover the corresponding decryption key. Under these circumstances, the encryption key can be made public and only the private decryption key kept secret.
- Tregere enn symetric-key
- (Public key, private key)

One-Way Functions:

- y=f(x) → lett
- Hva med å gå motsatt vei med å finne en x slik at y? → Vanskelig

Digital Signatures:

- **Hashing functions:**
  - MD5(Message Digest) (16-byte result)
  - SHA(Secure hash Algorithm) (20 byte result)



Trusted Platform Modules:

Protection Mechanisms

Protection Domains
Access Control Lists
Capabilities

Authentication

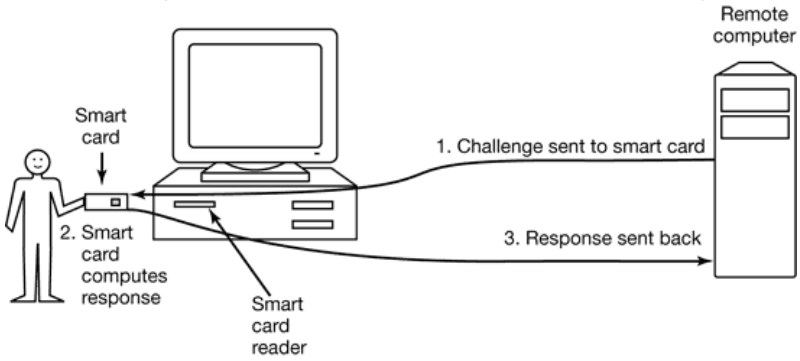
Identification:

1. Something the user knows.
2. Something the user has.
3. Something the user is.

Autentication using:

- **Password**
  - Inntrengere generer mange kombinasjoner (ofte navn på bruker) til de finner en kombinasjon av brukernavn og passord som finnes. Det finnes statistikk over hvaa som er de mest brukte passordene.
  - War dialer
  - IP, Ping, telnet (network attacks)
  - Default passowords (Mange skifter ikke default passord)
  - Viktig å sette krav til passord (min char, lowe and upper case, bokstaver + tall, unngå personlige relasjoner)
  - Engangspassord
  - Challenge-response (bruker spørsmål)

- **Physical Object:**
  - Smart card (dont need online connection to the bank)



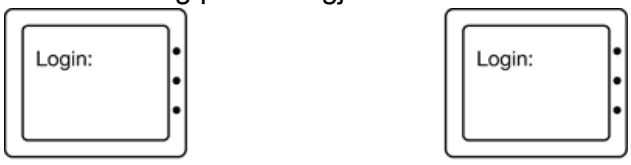
- **Using Biometrics:**
  - Fingerprint, voiceprint, etc.
  - Systemet må bestemme hvor nøyaktig signaturen skal være (bimetrics er vanskelig å måle nøyaktig!)

Insider Attacks

Logic Boms: Kodesnutt som gjør skade som blir fyrt av ved beskjed(feks hvis den ikke får sitt daglige passord feks så vil den fyres av neste dag). Man kan se på det som en faktisk bombe som går av og gjør skade. Ofte programmert inn av programmerer.

Trap Doors: Legge inn en kodesnutt som ikke ber om et passord (eller ikke bryr seg om hva passordet er) om brukernavnet er “abc” feks. → logges automatisk inn

Login Spoofing: Bruker tror at man skriver brukernavn og passord inn der man skal, men egentlig et det et skript/program som bare ser likt ut! Når bruker trykker enter vil programmet drepe seg selv etter den sensitive informasjonen er samla. Nå vil det orginale programmet vise seg. Deet ser helt likt ut og bruker vil nå tro at kanskje brukernavn/passord var feil og skriver nå inn brukernavn og passord igjen uten å tenke mer over det.

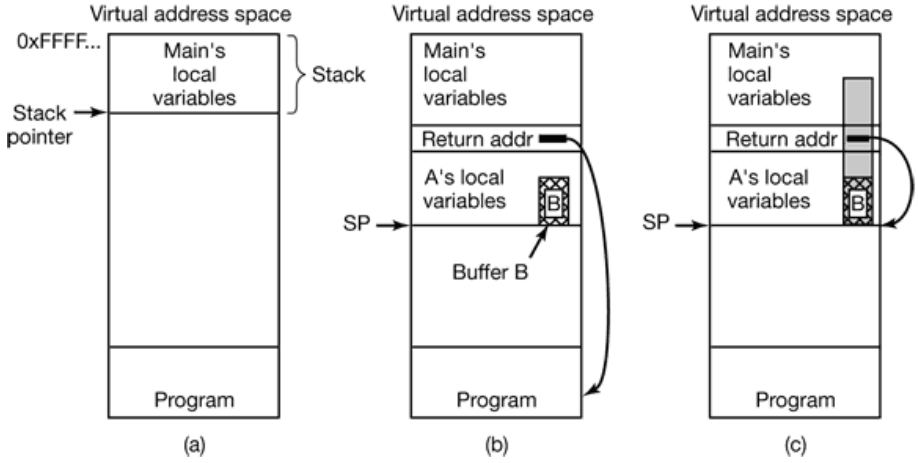


(a) Correct login screen. (b) Phony login screen.

Exploiting Code Bugs

Attacks:

- **Buffer Overflow:** C kompilator sjekker ikke for array bonunds



(a) Situation when the main program is running. (b) After the procedure *A* has been called. (c) Buffer overflow shown in gray.

- Format String:
- Return to libc:
- **Integer Overflow:**
- **Code Injection:** (Slette alle filer, sende passordfil til mail etc)
- **Priviliged Escalation:** Gi en bruker mer rettigheter enn den egentlig har.

Malware

Hvordan får du malware? Vi er dumme nok til å installere det selv!

Trojan Horses: Når man laster ned et kult spill eller program så får vi det ofte med malware med på kjøpet. Det følger ofte med i programmer som er gratis og nyttige. Når du starter det flotte programmet du lastet ned så er det mulig en funksjon som starter et annet program som kjører i bakgrunnen. Nå kan den kjøre hva den vil. Du åpnet døren og inviterte skurken inn! Slike gratispasasjerer kalles trojanske hester.

→ "... the beauty of the trojan horse attack is that it does not require the author of the trojan horse to break into the victims computer. Tge victim does all the work"

Viruses: et virus er et program som kan reprodusere seg selv med å ta sin egen kode å legge ved i et annet program. Kan se på det som vanlig smitte.
En person kan smitte en annen person → Et program kan smitte et annet program.

- **Companion virus:** does not actually infect a program, but gets to run when the program is supposed to run.
- **Executable virus:** viruses that infect executable programs . The simplest of this type just overwrites the executable program with itself.
- **Parasitic virus:**

- **Memory-resident virus:**
- **Boot secor virus:**
- **Device driver virus:**
- **Macro virus:**
- **Source code virus:**

Worms: are like viruses, but are self replicating. Bruker ordet virus for dette fenomenet også.

Spyware:

- Kan få spyware på samme måte som trojan horses eller bare som drive-by download (ved å besøke en infected webside), infected toolbar, activeX controls.
- **Tegn på spyware:**
  - Nettleserens forside er endret
  - Favoritter blir endret
  - Legger til toolbars
  - Endrer default mediaplayer
  - Endrer default nettleser
  - Legger til nye ikoner på skrivebordet
  - ….etc.

Rootkits:

- A rootkit is a program or set of programs that attempts to conceal its existence.
- **Types of rootkit:**
  - Firmware
  - Hypervisor
  - Kernel
  - Library
  - Application

Defenses

Firewalls: Bestemmer hvilke aplikasjoner som skal få komme inn (kan låse porter). Hva skal komme inn og hva skal komme ut av min PC.

Antivirus and antivirus techniques:

Code Signing:

Jailing:

Model-Based Intrusion Detection:

Encasulation Mobile Code:

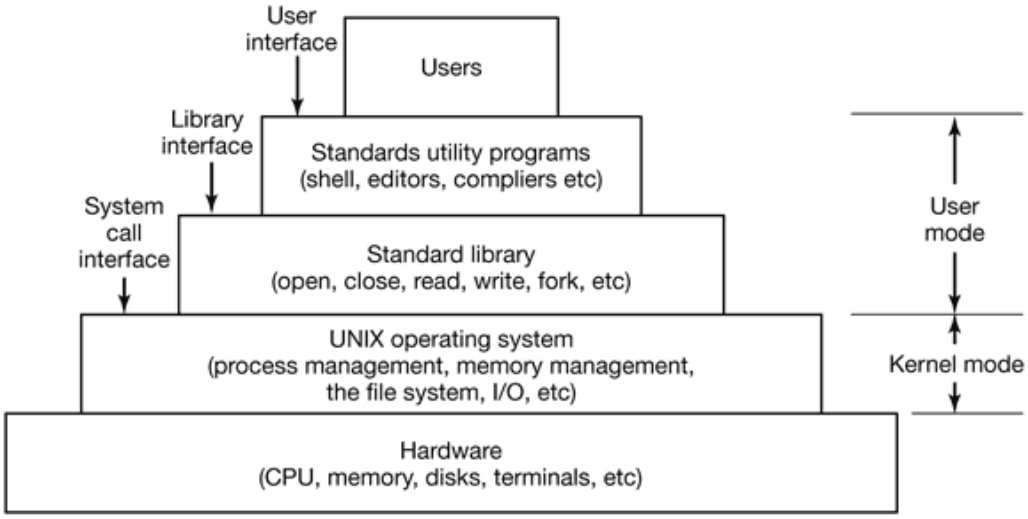
Java Security:

Kapittel10: “Case study: Linux”

10.2: Overview
10.3: Processes in Linux
10.4: Memory Management in Linux
10.6: The Linux File System

Overview:

Interfaces to Linux:



- For å at userprograms skal få tilgang til kernel utføres det systemkall
- Siden trap instruksjoner ikke kan skrives i C gjøres det bibliotekkall med en procedyre til hvert systemkall.

Linux Kernel:

- **Laveste nivå:** Interrupts og dispatcher
  - Dispatcher occurs when an interrupt happens. The low-level code here stops the running process, saves its state in the kernel process structures, and starts the approperiate driver.
  - Process dispatching also happens when the kernel completes some operations and it is time to start up a user process again.
- **Mellomnivå:** I/O, minne, prosess mgt
  - Alle linux drivere er klassifisert som enten: **character device** driver eller **block device driver** (Network device driver går som egen selv om det er char).
  - Forskjellen på de to klassene er at man kan bruke seek og random aksess på block devices and not on character devices.

- **Øverste nivå:** Systemkall

Processes:

Fundamental concepts:

- Each process runs a single program and initially has a single thread of control; in other words it has one program counter
- Linux allows a process to **create additional threads** once it starts executing
- Linux is a **multiprogramming system**
- **Daemons:** Background processes
    - cron daemon: vekkes opp hvert min for å se om det er arbeid å gjøre
    - Utfører generelle **periodiske akriviteter** som: Backup, vekking, kalenderavtaler med påminnelse, vekkeklokke osv. Dette er sovende prosesser som vekkes opp i perioder.
- The **fork** system call creates an exact copy of the original process. The forking process is called the **parent process**. The new process is called the **child process**.
- The parent and the child have their own private memory images.
- Open files are shared.
- **PID(Process Identifier):** Processes får et PID (prosessens navn)
- Når et **barn terminerer** så får forelderen barnets PID (på det grunnlag av at barnet kan ha barn igjen)
- En prosess kan bruke **getpid** for å finne sin egen PID.

Process communication:

- Prosesser kan kommunisere med hverandre ved å bruke en form for meldingssending.
- Det er mulig å lage en kanal mellom to prosesser → **pipes**
- Sort and head: all data sort writes go directly to head, instead of going to a file.
- Prosesser kan også kommunisere via **software interrupts** → A process can send what is called a signal to another process. Mottaker får noen valg:
    - Ignore signal
    - catch signal
    - la signal drepe prosess
- A process can only send signals to members of its process group.
- En prosess kan sende et signal til sin prosessgruppe ved et enkelt systemkall

Process management system calls in linux:

- Fork
- Waitpid
- exec
- cp

Implementation of processes in Linux:

- Every process has a user part that runs the user program
  → When a thread makes a system call, it traps to **kernal mode** and begin running in kernal context , with a different memory map and full access to all machine resources
- In kernal mode it is still the same thread but with its **own kernal stack and program counter**. These are **IMPORTANT** because a system call can **block part way through**. When blocked, the stack and PC are saved so the thread can go back in the same state in kernal mode and finish where it started.
- The Linux kernel internally represent processes as **tasks**, via the structure **task_struct**
- The kernal organizes all processes in a doubly **linked list of task structures.**
- The kernel maintains two key data structures related to processes, the **process table** and the **user structure**. The process table is resident all the time and contains information needed for all processes, even those that are not currently present in memory. The user structure is swapped or paged out when its associated process is not in memory, in order not to waste memory on information that is not needed.The
- **information in the process table** falls into the following broad categories:
    1. **Scheduling parameters.** Process priority, amount of CPU time consumed recently, amount of time spent sleeping recently. Together, these are used to determine which process to run next.
    2. **Memory image.** Pointers to the text, data, and stack segments, or, if paging is used, to their page tables. If the text segment is shared, the text pointer points to the shared text table. When the process is not in memory, information about how to find its parts on disk is here too.
    3. **Signals.** Masks showing which signals are being ignored, which are being caught, which are being temporarily blocked, and which are in the process of being delivered.
    4. **Miscellaneous.** Current process state, event being waited for, if any, time until alarm clock goes off, PID, PID of the parent process, and user and group identification.
- **The user structure** contains information that is not needed when the process is not physically in memory and runnable. For example, although it is possible for a process to be sent a signal while it is swapped out, it is not possible for it to read a file. For this reason, information about signals must be in the process table, so they are in memory all the time, even when the process is not present in memory. On the other hand, information about file descriptors can be kept in the user structure and brought in only when the process is in memory and runnable.
- **The information contained in the user structure** includes the following items:
    1. **Machine registers.** When a trap to the kernel occurs, the machine registers (including the floating-point ones, if used) are saved here.
    2. **System call state.** Information about the current system call, including the parameters, and results.
    3. **File descriptor table.** When a system call involving a file descriptor is invoked, the file descriptor is used as an index into this table to locate the in-core data structure (i-node) corresponding to this file.

4. **Accounting.** Pointer to a table that keeps track of the user and system CPU time used by the process. Some systems also maintain limits here on the amount of CPU time a process may use, the maximum size of its stack, the number of page frames it may consume, and other items.
5. **Kernel stack.** A fixed stack for use by the kernel part of the process.

- **Copy-on-write:** De jukser. Et barn får bare en “referranse” til forelderen sin page (som bare er read-only). Når da barnet skriver til denne siden vil en feil skje. Da vil data på forelderens page bli kopiert over i en ny page slik at barnet kan skrivet.
  → “..gjør ikke noe før man må prinsipp”.
  Grunnen til at dette er brukt er at det å kopiere minne er dyrt.

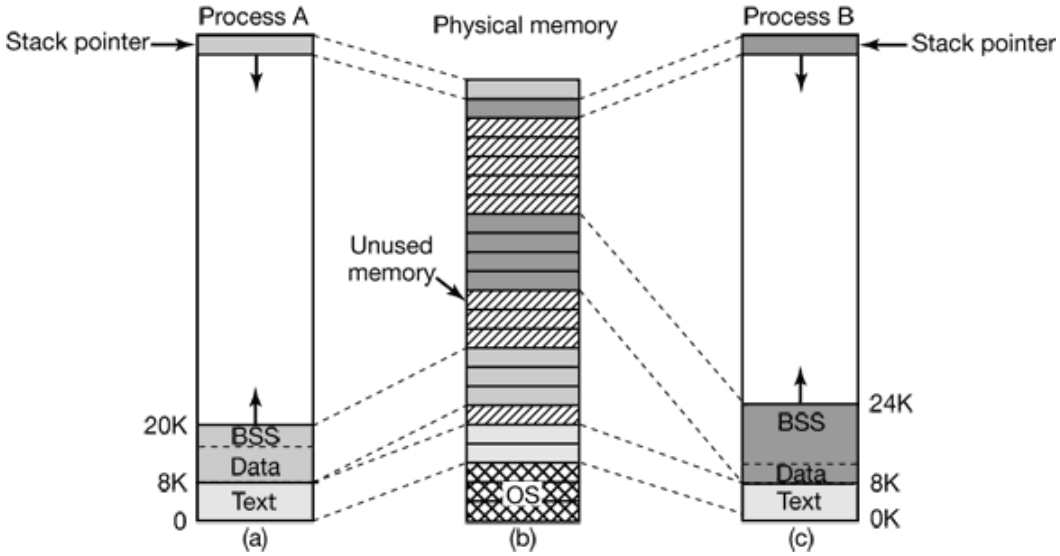
Implemetation of threads in Linux:

- The heart of the Linux implementation of threads is a new system call, clone, that is not present in any other version of UNIX. It is called as follows:
  pid = clone(function, stack_ptr, sharing_flags, arg);

Memory Management:

Fundamental concepts:

- Every Linux process has an address space logically consisting of three segments
  - **Text** (Programkode, Read only → Vokser ikke)
  - **Data** (program's variables, strings, arrays and other data)
    Har også en **BSS** (Block started by symbol).
    The to parts: the initialized data (data) and the uninitialized data(BSS)
  - **Stack**



(a) Process *A*'s virtual address space. (b) Physical memory. (c) Process *B*'s virtual address space.

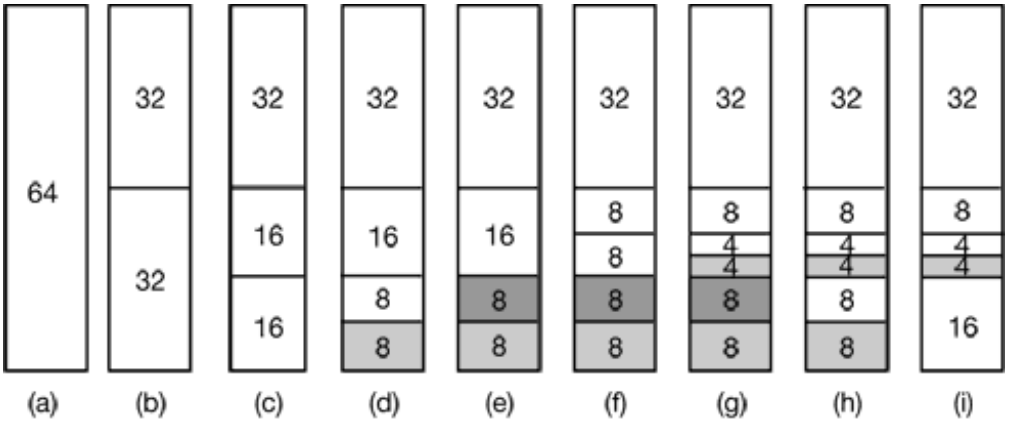
- Data og stack vokser. Dara vokser oppover og stack vokser nedover.

Memory management system calls in Linux:

System call	**Description**
s = brk(addr)	Change data segment size
a = mmap(addr, len, prot, flags, fd, offset)	Map a file in
s = unmap(addr, len)	Unmap a file

Implementation of memory management in Linux:

- **Physical memory management**
  - Linux deler opp i 3 minneområder:
    - ZONE_DMA (pages that can be used for DMA operations)
    - ZONE_NORMAL (Nomal, regulary mapped pages)
    - ZONE_HIGHMEM (pages with high-memory addresses, which are not permanently mapped.
  - Main memory in linux consists of three parts. The first two parts, the kernal and memory map, are pinned in memory. The rest of the memory is divided into page frames, each of which can contain a text,data or stack page, a page table or be on the free list.
  - page discriptor
  - zone descriptor (free areas).
  - node descriptor
  - Linux bruker 4-level page tables.
- **Memory Allocation Mechanisms**
  - **Buddy algorithm:** When request of memory comes in it:
    - round the memory needed (#pages) up to the power of two.
    - Deler i to helt til man finner en minnebit som er passe stor.
    - Linux bruker denne algoritmen + en array som peker på de resterende størrelsene slik at det blir lett å finne frem til ledige områder etter størrelse.
  - **Slab allocator**



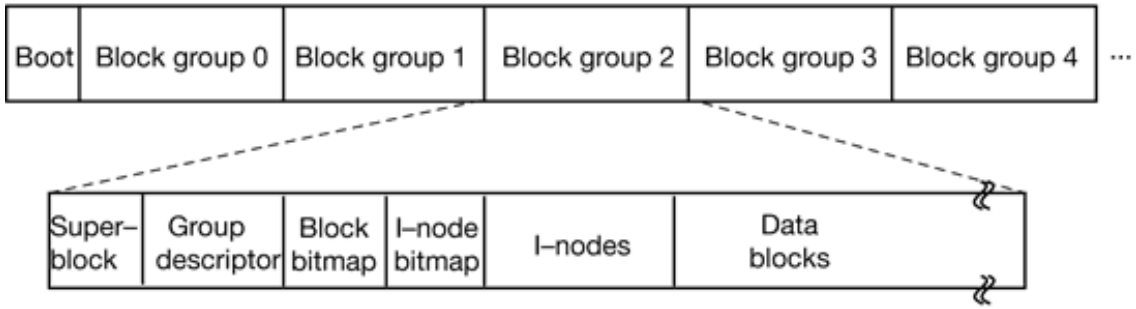
(The buddy algorithm)

- Virtual address space representation:
  -

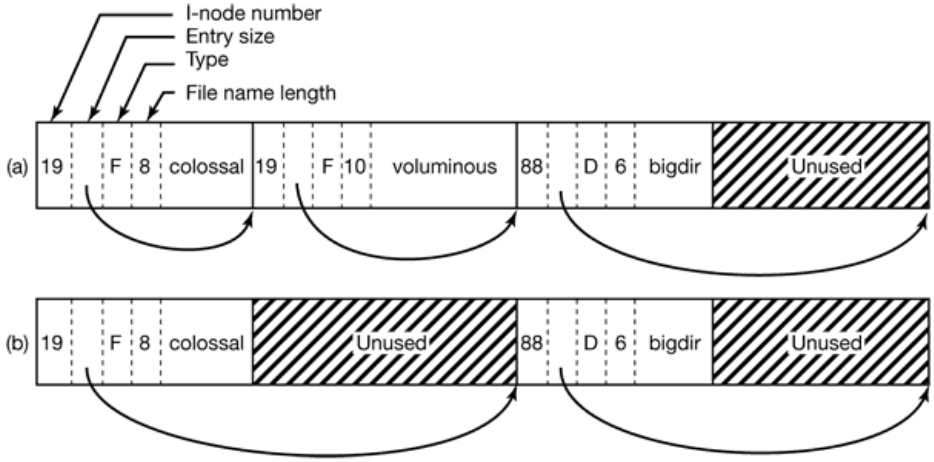
File System:

Implementation of Linux File system:

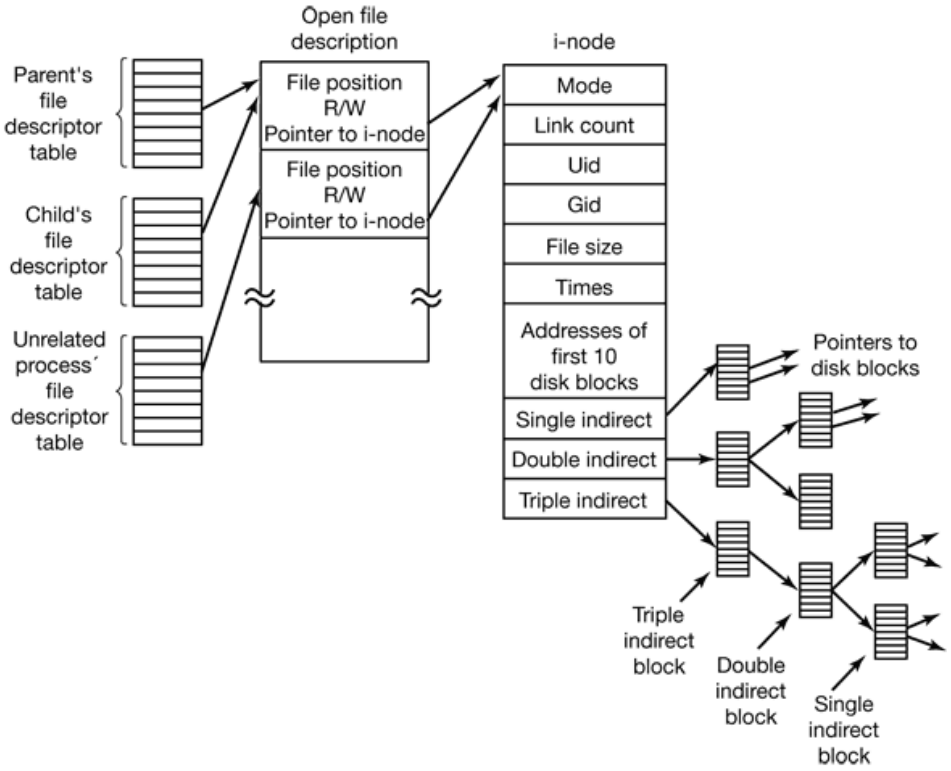
- The Linux virtual File system
  - File system **abstractions** supported by the Virtual firle system:
    - Superblock
    - Dentry
    - I-node
    - File
  - **Ext2 File system:**



    - The superblock contains information about the layout of the file system , #i-nodes, #disk blocks, start of the list of ffree disk blocks.
    - Group discriptor: Location of the bitmap, # free blocks in group, #i-nodes, #directories in the group
    - Bitmaps keeps track of free blocks and i-nodes
    - Each i-node is 128bytes and describes exactly one file
    - Datablocks is where the files and directories are stored.
    - Ext2 prøver å holde filer som hører sammen i samme datablock.



(a) A BSD directory with three files. (b) The same directory after the file *voluminous* has been removed.



(The relation between the file descriptor table, the open file description table, and the i-node table)

Oppsamling av div:

- Interrupt
- Trap
- Blokkerende systemkall: er et systemkall som ikke vil returnere før den har fullført sin jobb. Mens denne kjører må alle andre settes på vent til den er ferdig.
- Cache thrashing
- Meldingssending!!