*Hei, gjennom denne besvarelsen har jeg både brukt engels og norsk. Beklager dette. Det faller meg mest naturlig å svare på engelsk men ettersom oppgavene er på norsk ble det litt surr. Håper dette ikke skaper noe stress for dere.*

## 1: Utvikling av Operativsystemer

## a. Generelt

Vi bør aller først finne ut av hvordan maskinvaren er. Er det uniprosessor? Er det mulitkjerne? Er det multiprosessor? Er det et distribuert system? Dette vil påvirke nesten alt vi skal gjøre, inkludert om vi skal fokusere på tråder eller prosesser.

Hva er bruksområdet? Dette vil klargjøre om vi skal ha brukerfokus eller systemfokus, og om vi skal ha ytelsesfokus eller ikke.

Hvordan skal vi vurdere vårt OS? Hva er viktigst? Vi må bygge vårt OS slik at det er best mulig for den jobben det gjør og dette kan vi ikke uten å vite hvordan ”best” defineres i vårt kontekst.

Hvilken type data får vi? Generelt spørsmål som trengs for de fleste vurderinger.

Hvor stor mengde data får vi? Forteller oss noe om hvordan vi skal implementere lagre, VM, tidstyring, osv.

Hvordan er vektingen hvorav sikkerhet mot effektivitet? Dette er for å vite hvordan jeg skal bygge OSet og hvor mitt fokus skal ligge.

## b. Prosesser og tråder

Er det implementert memory protection?

Er det implementert en timer?

Hvilke interrupts har vi?

Kan vi ha priviligerte instruksjoner?

Er det lagt opp til instruction level parallelism?

Er aktivitetene isolerte eller skal de samarbeide? Prosesser fungerer bedre på isolerte enn tråder, mens tråder er bedre på samarbeid.

Hvor mange states er det lagt opp til i vår prosessormodell?

## c. Synkronisering:

Hvor viktig er det med konkurrens? Hvilke deler av minnet er kritisk? Dette må vi vite for å riktig skape konkurrens og riktige rammer for samhandlingen inni vår OS.

Hvor mye tillit skal vi ha til programmererne av applikasjonene? Hvis mye kan vi komme unna med bruk av semaforer, hvis ikke kan det være lurt å lage monitorer.

Trenger synkroniseringsmetoden å kunne brukes i både sentraliserte og distribuerte settinger. Isåfall må vi bruke meldinger.

## d. Lager

Hvordan er lageroppsettet? Generelt spørsmål som påvirker hvordan vi setter opp vårt OS.

Hvor store er lagrene? Generelt spørsmål som påvirker hvordan vi setter opp vårt OS.

Hvilke typer lagre har vi? Generelt spørsmål som påvirker hvordan vi setter opp vårt OS.

Er søking etter beste hull til datapakkene for ressurskrevende?

Gir den forhåndsdefinerte rammestørrelsen for mye intern fragmentering?

Har vi råd til tabelloverheaden av å bruke både segmentering og paging? Isåfall bruker vi begge fordi det beste fra begge.

Er oppdelingen av data i biter av samme størrelse: dette vil påvirke vårt valg av paging vs segmentering.

Skal vi legge opp tilpasset et programsynspunkt eller maskinsynspunkt? I henhold til Paging vs Segmentering.

Hvordan skal det gjøres med VM? Er det tilrettelagt for implementasjon av dette i HW? Har vi en TLB cache?

Hvor god hitrate kan vi regne med på vår cache (om vi har)? Ikke kjempenødvendig å vite men kan hjelpe optimalisering og estimering.

## e. tidsstyring av prosesser og tråder

Hva er bruksområdet? Vi må vite dette for å legge til rette for prioriteringer.

Hvordan vil bruksområdet ha innflytelse på hvilken selection function gir mest mening?

I vårt bruksområde, hvilken parameter er viktigst i valget av neste scheduled process?

Må prosessene utføres atomisk?

Er det multiprosessor eller multikjerne? Dette endrer våre tidstyringsvurderinger drastisk.

Hvordan er vektingen mellom *fairness* og effektivitet?

## f. I/O management

Hvilke I/O enheter har vi og hvordan fungerer disse? For eksempel er det ulike optimale algoritmer for ulike typer I/O.

Bruker vi mellomlagring eller nær-lagring?

Har vi DMA?

Har vi direkte kommunikasjon mellom DMA og lager?

Har vi egen buss dedikert til kommunikasjon mellom DMA og lager?

Hvordan er DMA implementert?

Alt dette trenger vi å vite for å drive effektivt.

# 2. Kompetanse om Operativsystemer

## a. Hvorfor må vi forstå en datamaskins avbruddssystem (interrupt system)?

Vi må forstå det for å kunne optimalisere effektivitet. Ved å kjenne avbruddsystemet kan vi bruke det til å øke processor utilization. Hvis vi ser på prosesser som skriver til en printer vil skrivingen av printeren gå mye tregere enn prosessoren. Uten avbrudd vil vi enten måtte vente på at den skal bli ferdig før vi fortsetter, eller så må vi begynne på en annen prosess uten å helt vite når printeren er ferdig og må derfor med jevne mellomrom sjekke. Dette er uansett ineffektivt. Ved å kjenne til avbruddsystemet kan vi i dette tilfellet fortsette med annet arbeid inntil vi blir fortalt igjennom avbrudd at printeren er ferdig. Og deretter gjenoppta arbeidet fra der vi forlot det.

Vi må forstå det for å kunne drive effektiv krasjunngåelse. Når noe går galt kan det sendes en interrupt med informasjon om hva årsaken er. Dermed kan vi drive damage control eller så kan vi unngå feilen helt ved å fikse deg som ikke stemmer.

## b. Hvordan vurderer du operativsystem-organiseringen i tidlig UNIX?

Det er dårlig tilpasset en moderne verden og er et produkt av utdatert teknologi. Dens kjerne støtter kun en type tidsstyringspolicy, en type filsystem og ett type kjørbart filformat. Det kan bare kjøre på en enkelt prosessor da den ikke har mulighet for lagerbeskyttelse ved konkurrens med flere brukere. Den er ikke passende for utvidelse og den ble heller ikke designet for det. Den er ikke effektivt utvidbar, og er heller ikke modular. Dette fører til at om vi skulle utvide den til moderne bruk ville det føre til en ineffektiv og ekstremt stor kjerne.

## c. Kan vi helt unngå aktiv bruk av CPU-kraft ifm. synkronisering av prosesser og tråder?

Jeg tolket spørsmålet som: Kan vi kjøre synkronisering uten bruk av CPU?

Nei, man må bruke CPU kraft for å synkronisere prosessorer og tråder. Uansett hvilken metode man bruker vil man måtte bruke CPU-kraft for oppdateringer og slikt.

## d. Er det viktig å ha både god maskinvarestøtte og god programvarestøtte ifm. lagerhåndtering?

Ja, det er viktig å ha gode lagerenheter og en passende blanding av disse. I tillegg må vi ha god støtte for et effektivt lagersystem. På HW siden er det viktig at vi har en god fordeling med Cache, main og sekundærminne. Denne blandingen vil være avhengig av bruksområdet om viktigheten av de forskjellige aspektene, slik som effektivitet eller treffsikkerhet. På SW siden så er det viktig med gode filesystemer, passende algoritmer og implementasjon av VM for å effektivisere bruk av lagerne og unngå fragmentering.

Ja, veldig viktig å utnytte prosessoren fullt mulig, noe som bare kan gjøres om vi har tilgjengelig prosessene som skal utføres i hovedminnet. Derfor må vi ha effektivt minnehåndtering for å sikre at til enhver tid er det størst sjanse for at vi ikke kaster bort prosessortid.

## e. Brukes de samme algoritmene for tidsstyring av prosesser/tråder på single-prosessorer og på multi-prosessorer?

Yes, when it comes to processes the situation is similar enough to use the samme algorithms for uniprocessors as for multiprocessors.

No. Threads behave differently and have other necessities when it comes to multiprocessors over uniprocessors. If a thread is blocked, it is beneficial for it to be restarted on a processor which has access to the cache of the previous used processor. If it is a multicore with communal cache within a chip, this is not an issue, however if not it is an issue, and we must adapt our algorithm for it. Hence, we cannot usually adapt the same algorithms for multiprocessing as for uniprocessors.

Both yes and no are valid answers here as the question is unspecific. ”prosesser/tråder” refers to two different situations, which supply different answers.

## f. Gjøres sideutbytting (page replacement) av cachet program-kode og cachet fil-data

## på samme måte?

They both use locality and wil operate with page replacement in a similar way

# 3. Ferdigheter

## a) Identifiser viktige kvalitative krav til et velfungerende operativsystem

**Data security**: Who can access what data?

**Restricted File system Access**: who can r/w what files?

**Concurrency**: Can allow multiple users and secure concurrency at once?

**Synchronization**: Are all processes and threads keeping up to date?

**Deadlock**: Can we ensure no interference from deadlocks, whether that is through avoidance, detection or prevention?

**Simplicity**: How easy is the OS to understand?

**File system**:

**Fair Scheduling:**

## b) Identifiser viktige kvantitative krav til et velfungerende operativsystem

**High efficiency**: How fast can it process jobs?

**High Reliability R(t):** How often is the OS usable?

**Lav Mean Time To Failure (MTTF)**: How long on average does it take for the OS to crash?

**High Processor Utilization**: How much of the time is processor in use?

**High Main Memory Utilization**: how much of main memory is used? Signifies little fragmentation.

**Few Page Fault Minimization**: Minimizing page faults will increase efficiency and processor utilization. It means less waiting for retrieving pages.

## c) Klarlegg større utfordringer ved implementering av prosess-/trådhåndtering

Usage of Kernel level threads vs usage of user-level threads

Scheduling: Which algorithm is optimal?

Mutual exclusion: How can we attain mutual exclusion and how can we do so without affecting performance.

## d) Klarlegg større utfordringer ved implementering av lagerhåndtering

Fragmentation, internal and external. This will cause poor utilization of memory and will increase the amount of page faults, which will in turn decrease efficiency.

Processes need to be assigned a logical contiguous part of memory. This is a problem whch can be remedied with VM.

Compaction is very expensive and we must find ways to minimize using this without it resulting in too much fragmentation.

## e) Analyser kritiske problemer som må løses for synkronisering av prosesser/tråder

Deadlock: We must solve deadlocks to secure completeness of system. This can be done by Avoidance, prevention or detection. Although prevention is subpar, we can avoid deadloccks by being mindful of resource allocation. Thsi is done either by process initiation denial or resource allocation denial. If we prefer a ”run and gun” type of approach, we can use deadlock detection and check whether one has occured and roll back.

Livelock: Several processes are changing because of a change somewhere else, which in turn triggers a change somewhere else. This is a lovelock which results in processing power used to change things in reaction to other changes which ends up doing no actual work.

Starvation: This is also something we must not let happen where a process is continually denied the necessary resources to do its thing. This is often avoided with good scheduling algorithms.

## f) Analyser kritiske problemer som må løses for tidsstyring av disker (disk scheduling)

Disks are incredibly slow relative to rest of the components in the computer. This can be somewhat remedied by using circular buffers.

If the disk has crashed, the OS will be tossing data at something which does not receive. Effectively throwing it in the garbage. This would be a critical error which some newer OS handle by checking whether there is a corruption before sending.

Data is often almost randomly spread over tracks. By using locality principle and good seeking algorithms, we can remedy this.