1.1 Explain the steps that an operating system has to perform when it executes a context switch between processes.

A context switch is essentially switching one running process to another process.

1. Stop running process
2. Save state of running process (registers, instruction pointer are saved on the stack or PCB)
3. Restore state of process 2
4. Resume operation on program counter for process 2

The different states a process can be in during a context switch is ready, running and blocked.

Skrevet av andre grupper på norsk:

1. Stop prosess 1, gjøre den om fra en "Running-tilstand" til "Ready-tilstand"
2. Lagre tilstanden til prosess 1 som instruksene som har blitt utført, hvor disse blir da lagret i registeret. Instruksjonspekerne vil da vite hvor i programmet som prosessen befinner seg i, fordi når prosessen skal starte om igjen så vil fortsette der den slapp.
3. Hente tilstanden til prosess 2
4. Kjøre prosess 2 , gjøre den om til "Ready-tilstand" til "Running-tilstand"

1.2 Explain the difference between processes and threads.

A process is a program under execution, while threads are lightweight processes that share the same resources as a process (such as address space). Multiple threads can share the resources of one process. What distinguishes threads from processes is that threads have their own state, stack, processor registers and program counter. Therefore, threads have no memory address switches and consequently thread context switching is cheaper than process context switching.

Skrevet av andre grupper på norsk:

Prosess er et program i utførelse, men en tråd er en utførelse av arbeidet i prosessen. Med andre ord er tråder miniprosesser som tilhører et overordnet prosess. Trådene deler de fleste ressursene i en prosess, og vil kjøre i samme omgivelse som prosessen, men tråder vil dele minne med hverandre. Men merk "Minne", det vil ikke inkludere stack, variabler, for å si bytes som er lagret lokalt i stacken, så har enhver tråd sin egen stack. Det trådene deler, er da heapen.

1.3 Explain why the performance difference between scheduling threads and scheduling processes makes very little difference when a large number of multithreaded programs runs concurrently on a server.

Threads have no memory address switches, which makes thread context switching cheaper than process context switches. Threads are therefore well suited for concurrent tasks running on a server. Servers have big time slices to reduce process context switching so they can benefit from having multiple threads running within a process. This utilizes cheap thread switches to achieve parallel execution of concurrent tasks within a process.

Skrevet av grupper på norsk:
Årsaken til ytelsesforskjellen ligger at man ikke kan unngå de ressurskrevende kontekstsvitsjende ved prosesser. Kontekstbytte mellom prosesser er dyrere enn et samlet antall kontekstbytter mellom tråder. Det skyldes av at trådene kjøres i parallell og kan utføre prosessen uavhengig av skeduleren. I tillegg til at trådene deler heaps så de kan derfor utføre andre operasjoner enn sine egne. Derfor vil ikke ytelsen ha en stor forskjell for et multitrådsystem siden kontekstbytte mellom prosesser er fremdeles dyrere enn et samlet antall tråder.

2.1 Explain the difference between preemptive and non-preemptive scheduling.
With preemptive scheduling it is possible to interrupt a running process so that another process with higher priority can execute first. In addition, it is also possible to preempt (let tasks wait for processing) if a task's time slice has been consumed. The preempted process continues at a later time in the same state it was previously in. With non-preemptive scheduling a new process can only start after the previous process has completely finished (terminated or in waiting state).

As such, preemptive scheduling is more flexible, but switching between processes is costly as it creates more context switches, which leads to overhead. In contrast, non-preemptive scheduling is less "costly" due to the less frequent switches. However, non-preemptive scheduling is more rigid which can create starvation down the queue. Meaning, if a high priority process enters the ready queue then it has to wait for its turn. Starvation can also happen in preemptive scheduling in the event that high-priority processes are frequently arriving in the ready queue - leaving the low-priority processes to wait their turn.

2.2 Explain how the two scheduling algorithms First In, First Out (FIFO) and Round Robin (RR) differ.
First In, First Out (FIFO) is a scheduling algorithm where tasks are processed in the order they arrive in the ready queue. Processes are therefore not being prioritized, which means that if one process should have higher priority, or it takes longer time to process - it simply has to wait its turn. Although it isn't necessary that all processes suffer from long waiting and finishing time since it may occur that some processes are lucky to finish their process within time. The problem is that it isn't evenly divided among processes for their finishing time, it is however more uneven. Consequently FIFO suffers from long waiting and finishing time, but it does have less frequent context switches which can be good for CPU-intensive jobs where we want to minimize overhead from switching.

Round Robin (RR) implements a FIFO queue where each process runs for a predetermined period of time (time slices) and is then preempted and put in the back of the queue. As such, all processes run in a cycle where each process gets a chance to run for a given time slice. This improves response time and therefore is better for interactivity. However, RR also creates more context switches and thus more overhead.

2. 3 Operating systems schedulers can be configured in very different ways. On workstations, they are configured to use very short time slices; many servers use much longer time slices, and some servers allow processes to run until they are finished. Explain why these different configurations exist. What are their advantages and disadvantages?
Different configurations of time slices on OS schedulers exist because different systems have different needs. For example, on workstations it is advantageous to use very short time slices to satisfy the users need for interactivity (good response time) and predictability (same performance over time). However, frequent context switches do create more overhead. Short time slices are therefore well suited for I/O bound processes.

Some servers, however, are usually configured with longer time slices to reduce process context switches which lead to less overhead and thus more throughput (stability and availability). Long time slices are therefore well suited for CPU bound processes. In fact, some servers want to reduce the context switching to almost zero by letting processes run until they are finished. A disadvantage of such non-preemptive scheduling is that processes down the queue can become starved.

In conclusion, short time slices are advantageous for I/O bound processes and long time slices are advantageous for CPU bound processes.

3.1 Hvilket scenario tilsvarer et avbrudd og hvilket scenario tilsvarer et unntak?
Det at hun blir avbrutt av pizzabudet er en asynkron hendelse, siden det er usikkert når avbrytelsen fra skolearbeidet kommer til å skje. Siden avbrytelsen kommer utenfra og den er asynkron, vil denne hendelsen kunne sammenliknes med et avbrudd.

Det at hun selv velger når pizzaen er ferdig, eller har en timer som går vil være en synkron hendelse. Det er fordi det er Alice selv som bestemmer når pizzaen skal ut av ovnen. Hendelsen er forventet, altså er den synkron. Siden denne hendelsen er synkron og håndtert av Alice selv vil dette kunne sammenlignes med ett unntak.

3.2 Reagerer prosessoren på begge hendelser på måter som er grunnleggende forskjellige eller hovedsaklig like?
Prosessoren behandler begge hendelsene hovedsakelig likt, når det gjelder byttingen av prosess, særlig traps og failures blir håndtert svært likt som interrupts. Hovedforskjellen er på hva som fører til byttingen, ved et avbrudd er det noe utenfor prosessen som kjører som fører til byttet, mens ved et unntak er det prosessen selv som fører til byttet.

3.3 Avbrudd resulterer vanligvis i tvungen overføring av kontroll fra den kjørende prosessen. For Alices del er hun tvunget til å ta en pause i studiene og passe på enten døren eller ovnen. Det er situasjoner der en prosess bør få lov til å fullføres. Hvordan håndteres avbrudd i slike situasjoner?
Vi antar at spørsmålet dreier seg om preemtive skedulerings algoritmer.

I en slik situasjon vil behandling av avbruddet bli kjørt etter at den kjørende prosessen blir satt til enten blocked, ready, eller blir ferdig. Deretter vil behandlingen av avbruddet skje som vanlig. Noen skedulerings algoritmer er preemptive og kan derfor prioritere prosessorer fremfor avbrudd, hvis prosessen har høy nok prioritering. Systemer som dette kan oppleve høy responstid, siden avbrudd fra tastatur, I/O (merker ikke denne på responstid) eller annet hardware blir behandlet senere etter avbruddet.

Hvis vi har en preemptiv skeduler, hvor prosess 1 har høyere prioritert enn avbruddet:
1. prosess 1 fullføres
2. avbruddet blir håndtert
3. prosess 2 begynner

Hvis vi har en preemptiv skeduler, hvor prosess 1 har lavere prioritert enn avbruddet:
1. prosess 1 blir stoppet
2. avbruddet blir håndtert
3. prosess 1 blir startet igjen

3.4 Unntak er delt inn i traps, faults og aborts. Beskriv scenarier som illustrerer de to sistnevnte underkategoriene.

Faults skjer ofte før en instruksjon blir utført, og lar maskinen blir tilbakestilt til en tidligere versjon som lar instruksjonene bli kjørt på nytt. For eksempel, Alice lager frossenpizza hjemme. Når hun skal til å åpne plasten ser hun at hun har tatt Tone sin pizza. Alice legger da tilbake Tone sin pizza i fryseren og henter sin egen og steker den istedenfor. Dermed er et kjennetegn ved faults er at handleren kan korrigere tilstanden.

Aborts skjer ofte alvorlige feil i systemet. Aborts er ofte destruktive, altså at arbeid går tapt og gir ofte lite presis informasjon om hva som skjedde. Et scenario som beskriver dette, kan være: Alice lager frossenpizza hjemme. Etter å ha satt pizzaen i ovnen setter hun seg ned og forbereder seg til morgendagens forelesning. Hun skvetter når brannalarmen går. Tone kom hjem å skrudde opp varmen fordi hun var sulten og ville ha pizza med en gang. Alice visste ikke om dette og vet dermed ikke hvorfor pizzaen ble brent før forventet. Resultatet er at pizzaen ble kastet.

2. Multitasking

Norsk

De fleste moderne operativsystemer støtter multitasking, det vil si evnen til å kjøre prosesser i omganger for å oppnå samtidighet.

- Forklar forskjellen mellom preemptiv og ikke-preemptiv skedulering.
- Forklar hvordan de to skeduleringssalgoritmene *First In, First Out* (FIFO) og *Round Robin* (RR) er forskjellige.
- Skedulerere for operativsystemer kan konfigureres på svært forskjellige måter. På arbeidsstasjoner er de konfigurert til å bruke svært korte tidsskiver; mange servere bruker mye lengre tidsskiver, og noen servere lar prosesser kjøre til de er ferdige. Forklar hvorfor disse forskjellige konfigurasjonene eksisterer. Hva er deres fordeler og ulemper?

Forklar forskjellen mellom preemptiv og ikke-preemptiv skedulering

Ved å forklare hvordan hver av disse skeduleringene fungerer, så ser vi allerede en forskjell. I en “preemptiv skeduler”, så vil en prosess som er av høyere prioritet, få støtte fra systemet til å kunne kjøre hvor en pågående “kjørende” prosess, vil bli avbrutt og satt på vent. Slikt fungerer det ikke i en “ikke-preemptiv skeduler“ hvor prosessene vil kjøre inntil de terminerer eller har brukt opp tiden sin. En kjørende prosess vil dermed ikke bli satt på vent av en høyere prosess som ligger i “Ready-queuen". Selvom det kan være prosesser som burde bli prioritert før alle andre, så må disse bli satt på vent på grunn av at en “ikke-preemptiv skedulering” ikke tar hensyn til prioriteter.

En annen forskjell er at i en preemptiv skedulering, så vil det forekomme context-switches som kan føre til overhead. Dette skyldes av at prosessene med høyere prioritet må veksle med en prosess som er i en kjørende tilstand. Dermed må en kjørende prosess “A”, bli satt på vent siden systemet bestemte at prosess “B” hadde høyere prioritet. Da skal prosess “B” få lov til å kjøre. Dette vil tilsvare en context-switch hvor vi må da lagre tilstanden til “A” slik at den får fortsette der den slapp eventuelt når prosess “B” er ferdig. Gjennom forelesningene så har vi lært at slike “switches” kan være kostbare og forårsake overhead. Context-switches vil ikke forekomme så ofte i en “ikke-preemptiv” skedulering ettersom prosessene som er satt på “running”, vil bli tillatt til å fullføre sin prosess eller kjøre så lenge tiden ikke er nådd. Dette vil skje uavhengig om det er andre prosesser med høyere prioritet som er satt på “ready”.

Tilslutt er preemptive skeduleringer “fleksible” ettersom prosesser som kommer inn i “Ready-køen” som er av høyere prioritetet, kan få lov til å kjøre med en gang. I motsetning til ikke-preemptive skeduleringer som ikke er fleksible ettersom prosessene for kjøre så lenge de terminerer.

Forklar hvordan de to skeduleringssalgoritmene “First In, First Out (FIFO)” og “Round Robin (RR)” er forskjellige

I en “FIFO”, så er skeduleren satt opp slik at den prosessen som ble satt inn først, er også den prosessen som skal utnytte CPU´en først. Hvis vi har en sekvens av prosesser i rekkefølgen “A,B,C”, så skal “A” få lov til å prosessere først, så “B” og tilslutt “C”. “RR” opptrer som en “FIFO” og fungerer slik som forrige setning. Forskjellen er at “RR” benytter “time-slices” som beskriver en tidsramme for hvor lenge en prosess får lov til å kjøre. Dermed vil en sekvens av “A,B,C” fremdeles kjøre i den rekkefølgen, men vi repeterer denne sekvensene gjentatte ganger på grunn av tidsrammen som er oppgitt. Hvis vi antar at enhver prosess i den overnevnte sekvensen krever lang tid å kjøre og at tidsrammen er kort, så må

enhver prosess bytte om tilstanden sin(Running til Ready) når de har brukt opp tidsrammen, settes tilbake til køen også kjøres om igjen når det er aktuelt for det. Dermed kan vi få mange “runder” hvor denne sekvensen gjentar seg flere ganger på grunn av tidsrammen. Dette er da forskjellen mellom “FIFO” og “RR”.

En annen forskjell er at noen prosesser kan ha lang ventetid i en “FIFO” i motsetning til “RR”. Hvis vi bruker sekvensen fra det forrige avsnittet, og antar at “A” har en lang prosesseringstid, imens “B” og “C” har meget korte tider, så må disse to vente en stund på grunn av “A”. Dette problemet unngår vi i “RR” ettersom “time-slicen” regulerer hvor lenge en prosess kan utnytte CPU´en. Dermed vil det være “rettferdig” fordelt og kortere ventetid sammenlignet med “FIFO”.

Tilslutt, kan en forskjell være å fortelle når man skal bruke den ene fremfor den andre. “RR” er f.eks bedre å bruke i I/O drevne prosesser siden slike prosesser ikke nødvendigvis bruker så lang tid på CPU´en sammenlignet med CPU-drevne prosesser. Imens “FIFO” vil være mer nyttig å bruke for CPU-drevne prosesser ettersom disse prosessene vil bruke en del tid på CPU´en og ved å ikke ha tidsregulering, så vil de få kjørt ferdig prosessen sin.

Skedulere for operativsystemer kan konfigureres på svært forskjellige måter. På arbeidsstasjoner er de konfigurert til å bruke svært korte tidsskiver; mange servere bruker mye lengre tidsskiver og noen servere lar prosesser kjøre til de er ferdige. Forklar hvordan disse konfigurasjonene eksisterer. Hva er deres fordeler og ulemper?

Hvorfor eksisterer slike konfigurasjoner?
Grunnen ligger i at ulike konfigurasjoner vil gi best ytelse for ulike kriterier. F.eks så er det viktig for interaktive systemer at I/O-drevne prosesser blir prioritert. Det er for å sørge for at brukeren skal få en god brukeropplevelse. Da burde nok en “preemptiv skedulering” være nyttig slik at et tastetrykk blir håndtert i rimelig god tid for å unngå forsinkelser for bruker. Hadde vi brukt FIFO, så kan det føre til dårlig brukeropplevelse for et interaktivt system. Vi

kan bruke sekvensen “A,B,C” hvor både “A” og “B” er CPU-drevne prosesser imens “C” er en I/O-bunden prosess. Siden CPU-drevne prosesser bruker lang tid for å kjøre, så må “C” vente lenge siden i “FIFO” så kjøres prosessene til de blir ferdig. Et annet eksempel er om vi har et multitaskingssystem. Da kan det være viktig å sette opp skeduleren slik at vi oppnår samtidighet blant prosesser.

Gjennom eksemplene som er blitt vist, så ser vi at det ikke finnes en skeduler som kan universelt brukes til alle systemer. Skeduleren må bygges i henhold til systemet som den blir benyttet på for å gi best ytelse for det gitte systemet. Dette er grunnen til hvorfor det eksisterer slike konfigurasjoner. Hvis vi ikke tar hensyn til systemet og målene vi ønsker å oppnå ved et system, så kan konsekvensene bli som følger: dårlig brukeropplevelse, mange context-switches som kan føre til delay, dårlig ressurshåndtering av CPU´en og mange andre problemer på grunn av at man ikke tar hensyn til systemet.

Fordeler og ulemper ved skeduler som er konfigurert med korte tidsskiver
Fordelene i en slik skeduler kan være at det blir rettferdig fordelt mellom prosessene når det gjelder kjøretid. Ulempen kan være at vi ikke tar hensyn til prosesser av høyere prioritet som burde få fullført hele sin prosess, fremfor å være begrenset til en gitt tid.

Fordeler og ulemper ved skeduler som er konfigurert med lange tidsskiver
Fordelene i en slik skeduler kan være at noen prosesser får terminert siden tiden er såpass lang. Ulempen kan være lang ventetid for prosesser som nødvendigvis ikke har lang kjøretid. Dette kan være I/O-drevne prosesser som må vente lenge på grunn av at det er en rekke CPU-drevne prosesser som er foran i køen.

Fordeler og ulemper ved skeduler som lar prosesser kjøre til de er ferdige
Fordelene i en slik skeduler kan være at vi unngår context-switches og overhead ettersom vi lar prosessene kjøre til de terminerer. Ulempene kan være at prosesser som er av høyere prioritet, må vente i “Ready-køen”.