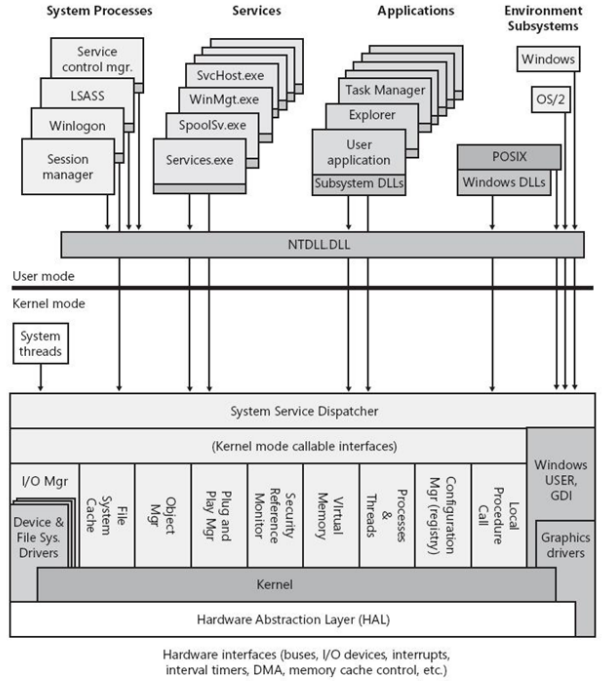
Examenvragen besturingssystemen

Hoofdstuk 1

1. Wat is het verschil tussen symmetrische en asymmetrische multiprocessing?
2. Wat moet je voorzien om op een Unix-systeem Windows applicaties te kunnen uitvoeren?
3. Bespreek hoe je op Windows een Unix-applicatie kan uitvoeren? Wat wordt in deze context bedoeld met een subsysteem?
4. Geef drie mogelijke ontwerpen van kernels. Bespreek bij elk hun voor- en nadelen en of ze nog gebruikt worden.
5. Gegeven onderstaande figuur:



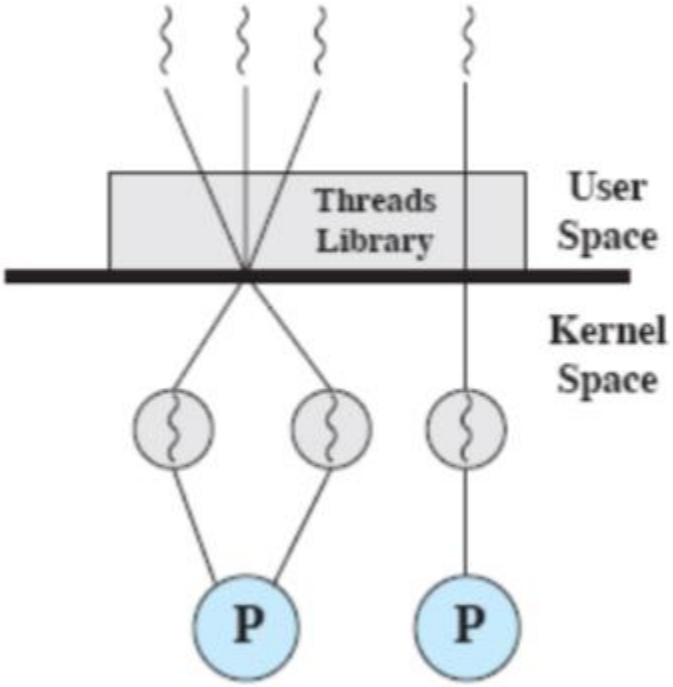
Geef van elke component in de “executive” aan wat de werking ervan is.

Hoofdstuk 2

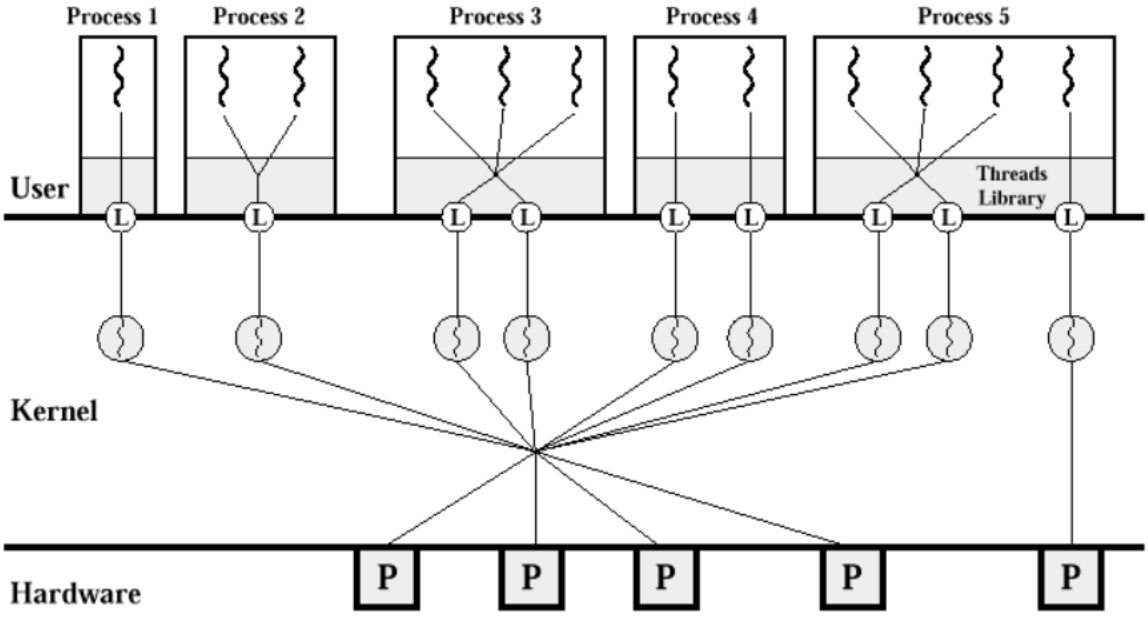
6. Wat is het verschil tussen een programma en een proces?
7. Waarom is het model met twee procestoestanden actief en niet actief niet interessant? Wat is het probleem dat je hier zal tegenkomen?
8. Geef alle toestanden waarin een thread zich kan bevinden? Bespreek wanneer een thread van de ene toestand in de andere zal terechtkomen.
9. Voor processen hebben we een model met 7 toestanden, dewelke? Teken het toestandsdiagram en geef aan hoe en wanneer er van toestand zal worden gewisseld.
10. Wat wordt er bedoeld met het procesbeeld? Geef aan hoe dit er uitziet en beschrijf ook wat er zich in elk deel bevindt. Het PCB mag je hier buiten beschouwing laten.
11. De info in het PCB kan je in drie categorieën onderverdelen. Dewelke? Bespreek ook wat er zich zoal in elk deel van het PCB bevindt.
12. Welke stappen moet het besturingssysteem ondernemen om een nieuw proces aan te maken?
13. Welke opportuniteiten kan het besturingssysteem gebruiken om van proces te wisselen?
14. Wat is het verschil tussen een synchrone en asynchrone interrupt?
15. Geef een aantal voorbeelden die aanleiding zullen geven tot het wisselen van proces. Geef een aantal voorbeelden die wellicht geen aanleiding zullen geven tot een proceswissel.
16. Hoe zal men bij een microkernel-architectuur een systeemaanroep afhandelen?
17. Hoe zal men bij een monolithisch kernelontwerp een systeemaanroep afhandelen?
18. Waarom hebben software interrupts een veel lagere prioriteit dan hardware interrupts?
19. Er wordt steeds gezegd dat wanneer een proces tegen een I/O-bewerking aanloopt, het proces geblokkeerd wordt. Hoe kan het besturingssysteem dat weten?
20. Bespreek de stappen bij het afhandelen van een interrupt wanneer de scheduler ervoor opteert om de uitvoering verder te zetten binnen het reeds actieve proces. Wat wordt er hier bedoeld met een moduswissel en een contextwissel?
21. Bespreek de stappen bij het afhandelen van een interrupt wanneer de scheduler ervoor opteert om de uitvoering **niet** verder te zetten binnen het reeds actieve proces. Welke stappen zijn nodig om een proceswissel door te voeren.
22. Wat zijn de nadelen van een procesloze kernel? Welke delen van een Unix- en een Windows kernel zijn procesloos?
23. Hoe wordt er van binnen een Unix besturingssysteem doorgaans van proces gewisseld? Hoe komt het dat dit vrij efficiënt verloopt?
24. Hoe wordt er binnen een microkernelgeoriënteerd besturingssysteem van proces gewisseld? Wat zijn hier de voor- en nadelen?
25. Wat is de definitie van een proces en de definitie van een thread?
26. Geef het procesbeeld van een multithreaded proces met drie threads. Welke delen worden er over de grenzen van een thread gedeeld.
27. Geef voor- en nadelen van multithreading. Welke zijn de mogelijke implementaties (enkel vernoemen volstaat)?
28. Wat zijn de voor- en nadelen met user level threading?
29. Wat zijn de voor- en nadelen van kernel level threading?
30. Wat is het verschil tussen coöperatieve- en preempted multitasking?

31. Geef het procestoestandsdiagram van een klassiek Unix besturingssysteem. Waarom is dit niet geschikt voor realtime-applicaties?
32. Uit welke drie delen bestaat een proces in een Windows besturingssysteem? Benoem ze en bespreek waarvoor ze dienen.
33. Geef het toestandsdiagram van een Windows thread? Bespreek de toestanden en de mogelijke overgangen.
34. Geef het toestandsdiagram van een besturingssysteem dat gebruikmaakt van user level threads en een lichtgewichtproces. Wat zijn de verschillende toestanden en de mogelijke overgangen?

35. Bespreek onderstaande figuur. Hoe worden de verschillende componenten aan elkaar gekoppeld.



36. Bespreek elke gegeven situatie en geef ook aan waar ze ideaal voor geschikt zijn, m.a.w. waar en wanneer zullen ze worden gebruikt.



Labo-gerichte vragen ter vervanging van hoofdstuk 3

37. Waarom krijg je bij het schrijven van data naar een filedescriptor die geopend werd met de O_SYNC-vlag een heel trage verwerkingssnelheid (2 redenen)?

38. Wat zijn de voor- en nadelen van (kernel/user) buffering? In de kernel wordt er default gebruikgemaakt van buffering en in het gebruikersprogramma stel je het zelf in. Bemerk dat wanneer je opeenvolgende schrijfopdrachten doet met slechts 1 byte, je per definitie geen buffering gebruikt.

39. Wanneer zal de systeemaanroep read resulteren in een geblokkeerd proces en wanneer niet. Idem voor een write-systeemaanroep. Geef nog een aantal andere systeemaanroepen die doorgaans een proces zullen blokkeren en dus bijgevolg een proceswissel zullen veroorzaken.

40. Geef een kort C-codefragment waarmee je een zombie-proces aanmaakt. Wanneer een proces een zombieproces wordt, welk deel van het proces houdt de kernel dan nog in het geheugen bij en waarom?

41. Na de systeemaanroep fork kan de uitvoering worden verdergezet in ofwel het kind ofwel de ouder. In welke van de twee zal de uitvoering hoogstwaarschijnlijk worden verdergezet en waarom? Wanneer zal dit niet het geval zijn?

42. Linux kent geen fork-systeemaanroep. Hoe komt het dat je hem wel kan gebruiken?

43. In welke situatie maak je gebruik van een named pipe en kan je dus geen unnamed pipe gebruiken?

44. Zijn POSIX-threads (of pthreads genoemd) kernel level threads of user level threads? Hoe kom je tot dit besluit?

Hoofdstuk 4

45. Aan welke vier randvoorwaarden moet ieder geheugenbeheersysteem voldoen?
46. Bespreek de werking van vaste partitionering (vaste grootte en verschillende grootte). Wat zijn de voor- en nadelen van dit systeem? Hoe zal het besturingssysteem procesbeelden gaan plaatsen in een systeem met vaste partitionering.
47. Bespreek de werking van dynamische partitionering. Wat zijn de voor- en nadelen? Welke zijn de verschillende plaatsingsalgoritmen en bespreek de werking en functie van elk algoritme.
48. Hoe gebeurt adresvertaling bij dynamische partitionering?
49. Bij dynamische partitionering kan je voor het geheugengebruik een bitmap of een gelinkte lijst bijhouden (maak een schets)? Hoe gebeurt dit en wat zijn de voor- en nadelen van beide systemen?
50. Bespreek de werking van paginering (zonder virtueel geheugen). Wat is het verschil tussen paginering en vaste partitionering? Hoe gebeurt de adresvertaling bij paginering (maak een schets)?

51. Bespreek de werking van segmentatie (zonder virtueel geheugen)? Wat is het verschil tussen dynamische partitionering en segmentatie? Waarom wordt dit model voor de gebruiker bewust zichtbaar wordt gehouden. Geef een voorbeeld waar je handig gebruik kan maken van segmenten. Hoe gebeurt de adresvertaling bij segmentatie (maak een schets)?
52. Bespreek de stappen die moeten ondernomen worden wanneer bij adresvertaling wordt vastgesteld dat een deel van het proces zich niet in het geheugen bevindt?
53. Wat zijn de drie voordelen van het gebruik van virtueel geheugen? Wat is het nadeel van het gebruik van virtueel geheugen?
54. Welke twee parameters moet het besturingssysteem in de gaten houden om te zien of er bij het gebruik van virtueel geheugen te veel dan wel te weinig paginafouten optreden? Wat wordt er bedoeld met “thrashing”? Hoe kan het besturingssysteem oordeelkundig inschatten welke pagina’s in de toekomst nodig zullen zijn en welke niet?
55. Welke extra zaken moeten er bij paginering in de paginatabel worden bijgehouden om paginering te kunnen doen in een systeem dat virtueel geheugen gebruikt. Hoe gebeurt de adresvertaling bij paginaring met virtueel geheugen (maak een schets)?
56. Welke parameters pleiten voor kleine paginagroottes en welke voor grote paginagroottes?

Oplossingen van vragen 37 tot en met 44

37. Waarom krijg je bij het schrijven van data naar een filedescriptor die geopend werd met de O_SYNC-vlag een heel trage verwerkingssnelheid (2 redenen)?
    Door die O_SYNC vlag zal iedere vorm van buffering worden uitgeschakeld. Iedere schrijfoperatie zal dus leiden tot een I/O-operatie.

    Wanneer je de vlag niet meegeeft zal een proceswissel niet steeds noodzakelijk zijn. Het proces roept via een software-interrupt de systeemaanroep aan, er treedt een modewissel en contextwissel op (cfr. de CPU-registers worden, in Linux althans, op de kernelstack bewaard) en de code van de systeemaanroep wordt in kernelmode uitgevoerd. Zonder de O_SYNC-vlag leidt dit veelal tot een zuivere geheugenoperatie door het wegschrijven van 1 of meerdere bytes van de user-buffer naar de kernel-buffer. De tijd die hiervoor nodig is is zeer beperkt en na de systeemaanroep is een proceswissel dus niet steeds aan de orde. Wanneer de kernelbuffer echter vol komt, moet er een I/O-operatie gestart worden en zal het gebruikersproces moeten worden geblokkeerd tot wanneer alle bytes uit de userbuffer werden gekopieerd.

    Bij het zetten van de O_SYNC-vlag heb je dus naast meer I/O-operaties ook steeds een extra proceswissel wat het vertragend effect nog versterkt. Iedere write-systeemaanroep wordt verplicht om een I/O-operatie te starten en het bijhorende proces te blokkeren tot wanneer de bijhorende I/O-interrupt het geblokkeerde proces deblokkeert.

38. Wat zijn de voor- en nadelen van (kernel/user) buffering? In de kernel wordt er default gebruikgemaakt van buffering en in het gebruikersprogramma stel je het zelf in. Bemerk dat

wanneer je opeenvolgende schrijfopdrachten doet met slechts 1 byte, je per definitie geen buffering gebruikt.

In usermode geen buffering gebruiken, cfr. 1 byte schrijfopdrachten dus, betekent niet dat er in kernelmode gebufferd wordt. Tenzij je uiteraard de O_SYNC vlag vermeld bij het openen van een filedescriptor.

Buffering beperkt het aantal I/O-opdrachten naar de schijf. De blokgroottes die naar schijf worden geschreven zijn niet bepalend voor de schijfprestaties, het aantal lees/schrijfoperaties per seconde (IOPS ook wel eye-ops uitgesproken, bij een klassieke magnetische disk een hondertal, bij een solid state schijf veel meer maar dan nog) daarentegen wel. Dus meer lees- of schrijfopdrachten beperkt de snelheid van het computersysteem in zijn geheel (i.e. veel meer processen in geblokkeerde toestand die aan het wachten zijn op een I/O-interrupt, ook het oplossen van paginafouten verloopt trager).

Buffering zorgt dat lees/schrijfopdrachten plaatsvinden op een geheugenbuffer en wanneer die dreigt vol te lopen zal het OS die moeten flushen. Er wordt dus van heel veel kleine I/O-opdrachten één grote I/O-opdracht gemaakt die dan daadwerkelijk naar schijf zal worden geschreven.

Het nadeel is wel dat je ten eerste geheugen zal moeten toekennen om die buffers te voorzien en twee dat er ook wel wat werkt kruipt in het constant in de gaten houden van de buffergrootte zodat die tijdig kan worden leeggemaakt (bij schrijven) of opgevuld (bij lezen). Buffering vereist ook complexe algoritmes die de buffers op juiste moment moeten aanvullen of leegmaken.

39. Wanneer zal de systeemaanroep read resulteren in een geblokkeerd proces en wanneer niet. Idem voor een write-systeemaanroep. Geef nog een aantal andere systeemaanroepen die doorgaans een proces zullen blokkeren en dus bijgevolg een proceswissel zullen veroorzaken.

Wanneer de buffer net moeten worden opgevuld, zal het OS het proces moeten blokkeren tot wanneer de achterliggende I/O-opdracht de buffer terug aangevuld heeft. Wanneer de buffer nog voldoende gevuld is zal het OS onmiddellijk de leesopdracht kunnen beantwoorden zonder te moeten wisselen van proces.

waitpid en bv. write (bij een volle buffer die moet geflusht worden) zijn voorbeelden van systeemaanroepen die een geblokkeerd proces en dus een proceswissel zullen veroorzaken.

40. Geef een kort C-codefragment waarmee je een zombie-proces aanmaakt. Wanneer een proces een zombieproces wordt, welk deel van het proces houdt de kernel dan nog in het geheugen bij en waarom?

```
int main(){
    int pid;
    if ( (pid=fork())<0){
            perror(argv[0]);
            exit(1);
    }
    else if (pid==0){
            //child
            exit(0);
    }
    //parent
    sleep(60);
    //niet wachten, geen waitpid dus
}
```

Wanneer een proces klaar is komt het in Unix in de status zombie terecht vooraleer het wordt afgebroken. Het proces kan echter uitsluitend worden afgebroken wanneer de parent zijn exit-status leest. Doet de parent dit niet, wordt het proces niet volledig beëindigd en blijft het in de toestand zombie. Als de parent zelf beëindigd wordt, wordt het kind naast een zombie ook een orphan dat geadopteerd wordt door het init-proces. Het init-proces zal de exit-status van het kind lezen waardoor vervolgens ook het kindproces kan worden ontmanteld.

De exit-status van een proces wordt in Linux bijgehouden in het PCB (process control block) dat dus zolang het in de status “zombie” vertoefd in het geheugen blijft.

Bij een daemon-proces zal je dus bij het niet correct lezen van de exit-statussen van de kinderen een ganse waslijst met zombie-processen maken die op zich 0% CPU tijd krijgen (dus geen probleem voor de scheduler) en ook 0% geheugen toegewezen krijgen. Iedere zombie neemt wel een plaats in de globale procestabel in beslag en bovendien blijft de volledige procesbesturingsinformatie achter in het geheugen! **Dit is dus niet onschuldig!**

41. Na de systeemaanroep fork kan de uitvoering worden verdergezet in ofwel het kind ofwel de ouder. In welke van de twee zal de uitvoering hoogstwaarschijnlijk worden verdergezet en waarom? Wanneer zal dit niet het geval zijn?

Wanneer je een systeemaanroep uitvoert komt na de uitvoering ervan in kernelmode de scheduler (ook in kernelmode) aan bod om te kijken of er door de interrupt-afhandeling geen meer prioritaire processen werden gedeblokkeerd. Wanneer er na de systeemaanroep geen andere I/O-interrupts werden afgehandeld die processen deblokkeren met een hogere prioriteit zal de uitvoering gewoon worden verdergezet in het proces dat aan zet was voor de onderbreking, de parent dus. Er is slechts één manier waarbij het kind als eerste aan bod zou kunnen komen en dat is dat bij het aanroepen van de systeemaanroep alle toegewezen tijd door de parent werd opgebruikt. Dan en enkel dan zal het kind als eerste aan bod komen.

42. Linux kent geen fork-systeemaanroep. Hoe komt het dat je hem wel kan gebruiken?

Linux gebruikt de glibc-POSIX API. Die staat in voor de vertaling van courante Unix-systeemaanroepen naar Linux-specifieke systeemaanroepen.

43. In welke situatie maak je gebruik van een named pipe en kan je dus geen unnamed pipe gebruiken?

Unamed pipes of anonieme pipes kan je uitsluitend gebruiken voor IPC (interprocescommunicatie) tussen processen met verwantschap (ouder, kind, broer, kleinkind,…). Bij processen zonder verwantschap moet je gebruikmaken van een named pipe, i.e. een object op het bestandssysteem dat je een naam moet geven (vandaar ook de naam).

44. Zijn POSIX-threads (of pthreads genoemd) kernel level threads of user level threads? Hoe kom je tot dit besluit?

Linux kent uitsluitend tasks en maakt eigenlijk geen onderscheid tussen threads en processen. Bij het gebruik van POSIX-threads merk je op dat er achter de schermen een clone-systeemaanroep wordt gebruikt en waar dus een nieuw proces mee wordt aangemaakt waarbij alles gedeeld wordt behalve dan de userstack. Ook wanneer je kijkt met de opdracht top -H zie je hier twee processen staan waarbij het dus eerder aanleunt bij de definitie van kernelthreads dan bij de definitie van user level threads. Ter info bij user level threads is het besturingssysteem niet op de hoogte van het gebruik van threads. De scheduler kan dus onmogelijk een user level thread voor uitvoering selecteren. Alle beheer van user level threads komt dus terecht bij de applicatie zelf. Het proces moet dus de nodige code bevatten om zelf te switchen tussen threads. Blokkeert één user level thread op een I/O-bewerking, dan blokkeert het volledige proces. Ook al kunnen andere threads binnen dat proces wel verder doen.