



BACHELOR TOEGEPASTE INFORMATICA

Processen

Computer Systems

2.0

Versie

Roel Standaert

Auteur(s)

HISTORIEK

Datum	Versie	Omschrijving	Auteur
28/09/2025	2.0	Update voor academiejaar 2025-2026.	Roel Standaert
29/09/2024	1.0	Eerste versie	Roel Standaert

INHOUD

6 PROCESSEN.....	4
6.1 INTRODUCTIE.....	4
6.2 PROCESSEN.....	5
6.3 PROCESS SCHEDULING.....	6
6.4 SYSTEM CALLS & INTERRUPTS	15

6 Processen



De student kent de basiswerking en -functies van een besturingssysteem

6.1 Introductie

Op een computersysteem staan er allerlei programma's: een browser, tekstverwerker, misschien een paar games, ... Deze programma's worden opgestart en beheerd door het besturingssysteem. In dit hoofdstuk gaan we bekijken hoe het besturingssysteem toelaat om meerdere programma's (tegelijk) te draaien (process scheduling) en hoe het besturingssysteem deze processen toelaat om dingen te doen zoals input/output (system calls).

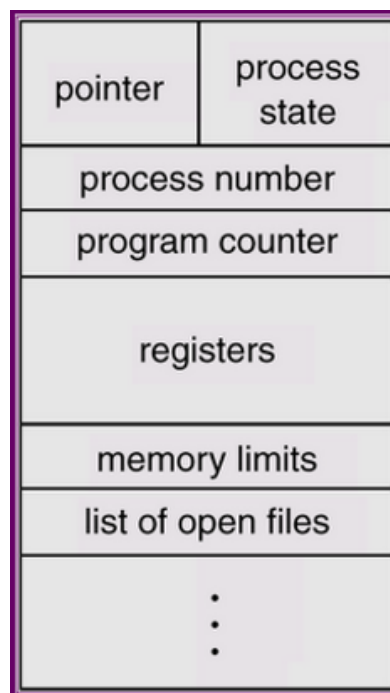
6.2 Processen

De programma's op een computersysteem draaien natuurlijk niet allemaal constant. Van zodra je een programma start wordt het een proces. Een proces is dus een programma in uitvoering.

Het besturingssysteem houdt voor een proces allerlei informatie bij, zoals:

- Waar het proces zich op dat moment in de code van het programma bevindt (program counter)
- De bestanden die op dat moment door het proces gebruikt worden (open files)
- Welk geheugen wordt voor het programma gebruikt?
- De status van het proces (draaiend, wachtend op input, ...)

Figuur 6-1 illustreert de toestand van een proces.



Figuur 6-1: Een schematische voorstelling van de process state.

6.2.1 Processen op Linux

Om even te illustreren welke procesinformatie je besturingssysteem zoal bijhoudt kunnen we eens gaan kijken naar het proc-bestandssysteem. Dit is een speciaal bestandssysteem (of *filesystem*) dat je toegang verschaft tot de informatie die de Linux kernel heeft over processen. Dit speciale bestandssysteem is toegankelijk

onder het pad `/proc`. Probeer eens `ls /proc` uit te voeren in de terminal. Je zal zien dat hier allerlei genummerde mappen in staan. Deze mappen omschrijven elk de toestand van een proces op jouw systeem. Het nummer verwijst naar de “process id” van het proces.

Bekijk nu eens de map met info over je huidige shell:¹

```
ls -la /proc/$BASHPID
```

Je zal hier een heleboel bestanden zien staan. Deze bestanden staan niet ergens op jouw harde schijf of SSD, maar zijn een soort virtuele bestanden die toegang geven tot de informatie die de Linux kernel heeft over dit proces.

Enkele voorbeelden:


- `/proc/$BASHPID/exe` is een link naar het programma (`/usr/bin/bash`).
- `/proc/$BASHPID/fd` is een map die alle open *file descriptors* bevat. Dit zijn verwijzingen naar alle bestanden die het proces op dit moment gebruikt.
- `/proc/$BASHPID/limits` bevat alle limieten die de Linux kernel op dit moment heeft ingesteld op dit proces (bv. maximaal geheugengebruik).

6.3 Process scheduling

De computers van weleer hadden geen besturingssysteem. De code van programma's werd vroeger typisch op ponskaarten gezet. Dit zijn kaarten met gaatjes in waarop de binaire instructies stonden die door de processor moesten worden uitgevoerd. Een gaatje kwam overeen met een 1, geen gaatje een 0. De menselijke operator was verantwoordelijk om deze programma's uit te voeren: letterlijk de ponskaarten in de computer steken.

Deze menselijke operator kreeg niet één programma om uit te voeren, maar veel verschillende programma's. Eén computer werd bijvoorbeeld gedeeld door een heel departement van een universiteit. Niemand had een eigen computer, dus alle

¹ BASHPID is een variabele die verwijst naar de *process id* van de huidige bash shell. Met het dollarteken (\$) wordt deze waarde ingevoerd. Als onze bash shell dus *process id* 10 heeft, dan wordt het commando `ls -la /proc/10`.



professoren en studenten moesten hun programma's aan de operator geven om ze uit te voeren. Deze menselijk operator moest dan beslissen in welke volgorde deze programma's uitgevoerd moesten worden. Een programma van een professor zou bijvoorbeeld meer voorrang genieten dan dat van een student.

Tegenwoordig wordt deze taak vervuld door het besturingssysteem. Jouw computer heeft een beperkt aantal processorkernen en één van de taken van het besturingssysteem is bepalen wanneer welk proces de kans krijgt om op een bepaalde processorkern uitgevoerd te worden. Dat gedeelte van het besturingssysteem noemen we de **process scheduler**.

In deze sectie gaan we drie verschillende strategieën (algoritmes) beschrijven die kunnen gebruikt worden om processen te *schedulen*. Om het eenvoudig te houden gaan we ervan uitgaan dat er slechts één processorkern is. Als er meerdere cores zijn, is de *scheduler* natuurlijk iets complexer, maar de onderliggende principes blijven hetzelfde.

6.3.1 Preëemptief / niet-preëemptief

Bij *scheduling*-algoritmes kunnen we het onderscheid maken tussen preëemptieve en niet-preëemptieve methodes:

- **Preëemptieve** algoritmes kunnen een draaiend proces onderbreken. Dit bijvoorbeeld om een ander proces de kans te geven om even uitgevoerd te worden.
- **Niet-preëemptieve** algoritmes zullen draaiende processen niet onderbreken. Het is volledig aan het proces zelf om te beslissen wanneer het even niet wil uitvoeren.

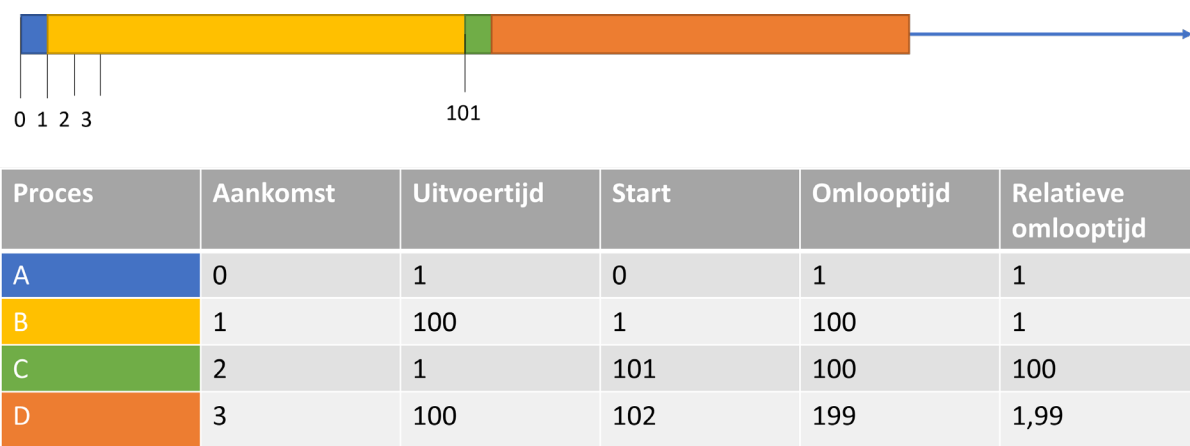
Sommige oude besturingssystemen, zoals Windows voor Windows 95, gebruikten niet-preëemptieve *scheduling*-algoritmes. Tegenwoordig gebruikt bijna elk modern besturingssysteem een preëemptief *scheduling*-algoritme.

6.3.2 First Come, First Serve

Het eenvoudigste scheduling-algoritme is **First Come, First Serve** (FCFS). FCFS behandelt processen die wachten om uitgevoerd te worden zoals de wachtrij in een supermarkt met maar één kassa: de eerste die toekomt mag eerst afrekenen. FCFS voert dus eerst het eerste wachtende proces uit tot die “klaar” is (omdat het proces stopt of omdat het proces zichzelf pauzeert, bijvoorbeeld omdat het wacht op input van de gebruiker).

FCFS is een voorbeeld van een niet-preëmptief algoritme: de *scheduler* gaat niet zelf beslissen om een proces te pauzeren. Het proces wordt dus volledig uitgevoerd (of tot dat proces zelf pauzeert), waarna het volgende proces wordt uitgevoerd, enzovoort.

Voorbeeld van FCFS



Figuur 6-2: FCFS scheduling

Figuur 6-2 illustreert FCFS voor 4 processen. Bovenaan zie je een tijdslijn die toont wanneer welk proces uitvoert. Daaronder vind je een tabel waar voor elk proces wordt gegeven:

- De **aankomsttijd** (het tijdstip waarop het proces “aankomt” bij de scheduler). Dit is dus het vroegst mogelijke tijdstip dat een proces zou kunnen starten met uitvoeren. Proces A komt dus aan op tijdstip 0, proces B op tijdstip 1, enzovoort.
- De **uitvoertijd**: hoe lang dit proces moet uitvoeren. Proces A moet 1 tijdseenheid uitvoeren, proces B 100 tijdseenheden, enzovoort.

- **Start:** het moment waarop het proces effectief begint met uitvoeren.
- **Omlooptijd:** dit is de tijd die verstreken is tussen de aankomsttijd en het einde van het uitvoeren van een proces. De totale wachttijd van een proces is dus gelijk aan de omlooptijd min de uitvoertijd.
- **Relatieve omlooptijd:** dit is de omlooptijd van een proces, relatief ten opzichte van de uitvoertijd (omlooptijd gedeeld door uitvoertijd).

FCFS *schedulet* deze processen als volgt:

1. Je ziet dat proces A aankomt op proces 0, meteen kan beginnen met uitvoeren (er zijn dan nog geen andere wachtende processen). Omdat dit proces maar 1 tijdseenheid uitvoert, is het ook klaar met uitvoeren op tijdstip 1. Omdat proces A niet moest wachten om te kunnen beginnen met uitvoeren is de omlooptijd dus ook gelijk aan de uitvoertijd: 1 tijdseenheid. Dit geeft de best mogelijke relatieve omlooptijd: 1.
2. Proces B komt dan aan op tijdstip 1 en heeft 100 tijdseenheden nodig om uit te voeren. Het kan meteen beginnen met uitvoeren en is klaar op tijdstip 101. Wederom moest proces B niet wachten, dus hebben we hier ook weer de best mogelijke relatieve omlooptijd: 1.
3. Proces C komt aan op tijdstip 2, terwijl proces B aan het uitvoeren is. Proces C zal dus volgens FCFS moeten wachten tot proces B helemaal klaar is. Pas op tijdstip 101 kan het beginnen met uitvoeren. De uitvoertijd van proces C is slechts 1 tijdseenheid, dus op tijdstip 102 is proces C klaar. De korte uitvoertijd (1 tijdseenheid) in combinatie met de lange wachttijd (99 tijdseenheden) geeft een relatief hoge omlooptijd van 100 tijdseenheden en ook een relatieve omlooptijd van 100.
4. Proces D komt aan op tijdstip 3 en moet wachten tot zowel proces B als C klaar zijn met uitvoeren. Op tijdstip 102 kan het proces dan starten en het is klaar op tijdstip 202. Dit geeft een omlooptijd van $202 - 3 = 199$ en een relatieve omlooptijd van $199/100 = 1,99$.

Voordelen van FCFS

First Come, First Serve is een goed algoritme voor processen die in lange CPU *bursts* worden uitgevoerd. Dat zijn de typische “*batch*”-processen die geen interactie vereisen, bijvoorbeeld een video die geëxporteerd wordt uit een videobewerkingsprogramma.

Nadelen van FCFS

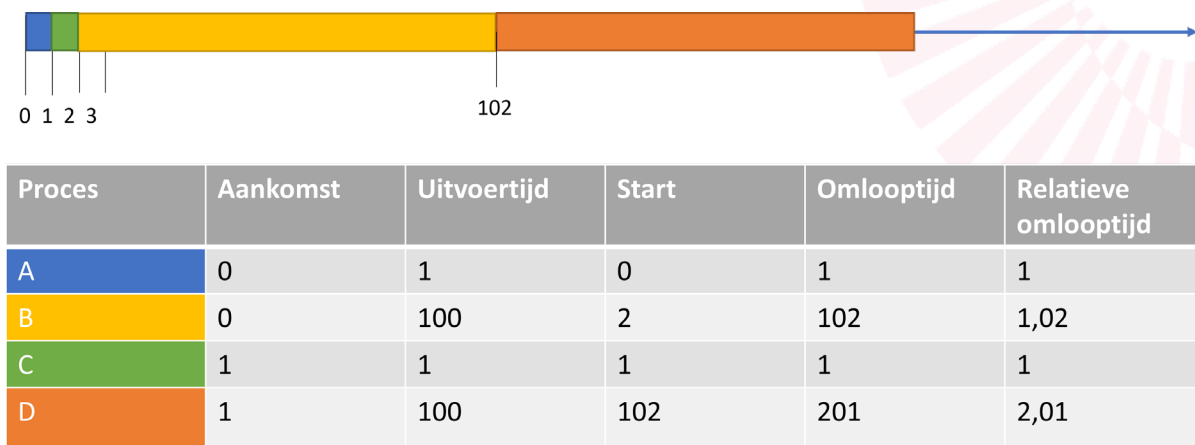
Het nadeel is dat korte processen die later aankomen vaak relatief lang moeten wachten, zoals proces C in ons voorbeeld. Om de kassa-vergelijking er weer bij te halen: het is zoals achter iemand staan met een volle winkelkar terwijl je eigenlijk maar een paar kleine dingen wil kopen.

Ook moet een proces dat vrijwillig even pauzeert (bijvoorbeeld omdat het wacht op input) telkens weer helemaal achteraan aanschuiven en kan het zo achter andere processen belanden met langere CPU *bursts*.

6.3.3 Shortest Job First

We hebben gezien dat FCFS ervoor zorgt dat heel korte processen heel lang moeten wachten om uitgevoerd te worden. Shortest Job First (SJF) *scheduling* geeft voorrang aan kortere processen. Het kortste proces wordt eerst uitgevoerd, gevolgd door het op een na kortste proces, enzovoort.

Voorbeeld van SJF



Figuur 6-3: SJF Scheduling

Figuur 6-3 toont een voorbeeld van SJF. SJF zal deze vier processen *schedulen* als volgt:

1. Proces A en B komen gelijktijdig aan. De uitvoertijd van proces A (1) is echter korter dan B (100), dus proces A wordt geselecteerd om uit te voeren. Proces A voert dus 1 tijdseenheid uit om te eindigen op tijdstip 1, met een relatieve omlooptijd van 1.
2. Op tijdstip 1 komen proces C en D aan. C heeft de kortste uitvoertijd, dus wordt als volgende uitgevoerd. C eindigt op tijdstip 2, met een relatieve omlooptijd van 1.
3. Nu blijven B en D over. Deze processen hebben dezelfde uitvoertijd (100). Als *tiebreaker* zullen we het proces dat het eerste aankwam uitvoeren (proces B). Proces B eindigt op tijdstip 102, wat zorgt voor een omlooptijd van 102 of een relatieve omlooptijd van 1,02.
4. Tenslotte kan D uitvoeren. Dit proces eindigt op tijdstip 202 en heeft dus een omlooptijd van 201 tijdseenheden, of een relatieve omlooptijd van 2,01.

Voordelen van SJF

Het voordeel van SJF is dat korte processen heel snel afgehandeld zullen worden, en dat dus daardoor de gemiddelde omlooptijd laag wordt gehouden.

Nadelen van SJF

Het nadeel is dat processen met langere *bursts* zullen worden uitgesteld. Als een proces 1000 tijdseenheden moet uitvoeren, maar er komt telkens een proces voor dat slechts 10 tijdseenheden nodig heeft dan wordt het lange proces steeds uitgesteld.

Dit fenomeen, een proces dat nooit aan de beurt komt, wordt **starvation** genoemd. Om dit tegen te gaan kan er **aging** worden toegepast. De *scheduler* houdt dan rekening met hoe lang processen al aan het wachten zijn.²

² Aging kan op verschillende manieren worden geïmplementeerd, maar daar gaan we niet dieper op in.

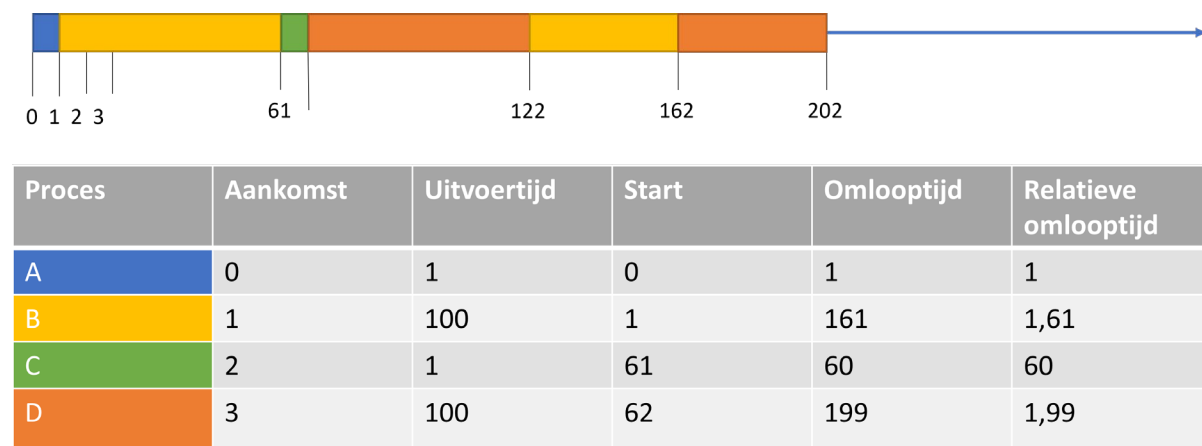
6.3.4 Round Robin

We kunnen ook voorkomen dat processen lang moeten wachten om uit te voeren door te beperken hoe lang elk proces telkens kan uitvoeren. Deze korte uitvoeringsperiode noemt men een **tijdsquantum** en de algoritmes die hiervan gebruik maken noemt men **preëemptieve** algoritmes.

Het **Round Robin**-algoritme (RR) is een voorbeeld van een preëemptief algoritme.

Het werkt net zoals FCFS maar dan preëemptief. Een proces krijgt telkens een bepaalde tijd om uit te voeren (het voorgenoemde tijdsquantum) en wordt daarna onderbroken en weer achteraan in de wachtrij geplaatst.

Voorbeeld van RR



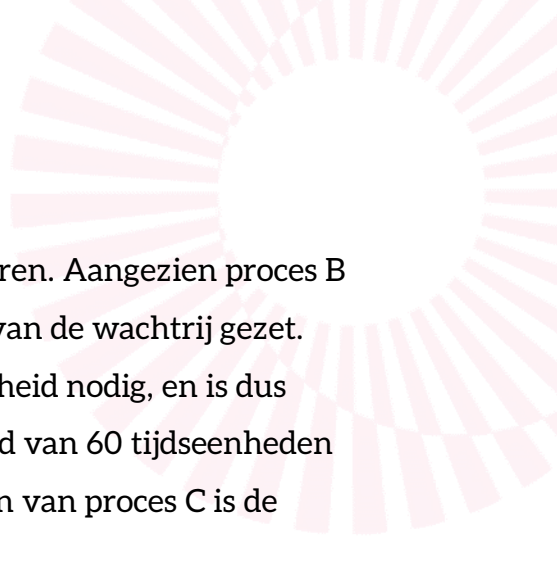
Figuur 6-4: RR scheduling

Figuur 6-4 illustreert *Round Robin scheduling* met een tijdsquantum van 60 tijdseenheden. RR zal deze 4 processen *schedulen* als volgt:

1. Proces A komt als eerste aan en moet maar 1 tijdseenheid uitvoeren. Dit is minder dan de tijdsquantum (60) dus proces A wordt meteen volledig uitgevoerd en eindigt op tijdstip 1.
2. Proces B komt aan op tijdstip 1 en heeft een uitvoertijd van 100. Dit is groter dan de tijdsquantum (60), dus proces B mag maar 60 tijdseenheden uitvoeren voor het terug in de wachtrij geplaatst wordt. Ondertussen zijn ook C en D aangekomen.

De wachtrij is nu dus:

- a. Proces C

- 
- b. Proces D
 - c. Proces B met nog 40 tijdseenheden uit te voeren. Aangezien proces B net heeft uitgevoerd wordt het op het einde van de wachtrij gezet.
3. Proces C mag nu uitvoeren en heeft maar 1 tijdseenheid nodig, en is dus klaar op tijdstip 62. Dat geeft proces C een omlooptijd van 60 tijdseenheden en een relatieve omlooptijd van 60. Na het uitvoeren van proces C is de wachtrij:
- a. Proces D
 - b. Proces B met nog 40 tijdseenheden uit te voeren.
4. Proces D mag nu uitvoeren, maar heeft meer dan 60 tijdseenheden nodig. Na 60 tijdseenheden (op tijdstip 122) zal proces D dus worden gepauzeerd. De wachtrij is nu:
- a. Proces B met nog 40 tijdseenheden uit te voeren.
 - b. Proces D met nog 40 tijdseenheden uit te voeren.
5. Proces B mag nu uitvoeren, en is na 40 tijdseenheden klaar (tijdstip 162). De omlooptijd van proces B is dus $162 - 1 = 161$ tijdseenheden, met een relatieve omlooptijd van 1, 61. Nu staat slechts proces D nog in de wachtrij.
6. Proces D voert 40 tijdseenheden uit tot tijdstip 202. De omlooptijd van proces D is dus $202 - 3 = 199$, met een relatieve omlooptijd van 1, 99.

Voordelen van RR

Round Robin leidt tot minder lange wachttijden dan FCFS, aangezien de tijd tot een proces mag beginnen uitvoeren beperkt wordt. Je kan zien dat het korte proces C in ons voorbeeld nu sneller aan de beurt komt, en sneller klaar is dan bij FCFS.

Er kan ook geen *starvation* optreden bij RR, aangezien een proces dat op plaats N staat maar maximaal $(N - 1) \times Q$ tijdseenheden (met Q het tijdsquantum) zal moeten wachten om te mogen uitvoeren.

Nadelen van RR

Wisselen tussen processen, genaamd *context switch*, is echter niet gratis. Het besturingssysteem zal bepaalde informatie over het proces dat gestopt wordt moeten wegschrijven naar het werkgeheugen en het nieuwe proces telkens moeten

inladen. Dat introduceert wat extra *overhead*. Hoe korter de tijdsquantum, hoe meer die overhead zal doorwegen.

Een ander nadeel is dat processen die zichzelf vaak vrijwillig onderbreken (zoals bijvoorbeeld om te wachten op input) ook weer telkens achteraan moeten aanschuiven. Hun wachttijd mag dan wel beperkt zijn tot het aantal processen ervoor maal het tijdsquantum, maar als er genoeg andere processen zijn kunnen dit soort processen, zoals tekstverwerkers *laggy* aanvoelen: het duurt een tijdje voor een input van de gebruiker verwerkt kan worden.

6.3.5 Scheduling in de praktijk

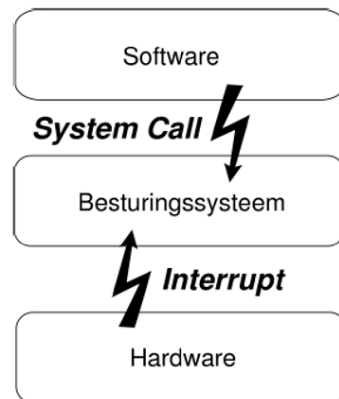
In de realiteit worden bovenstaande algoritmes niet op zich gebruikt, maar wordt er verder op voortgebouwd en worden verschillende *scheduling* algoritmes gecombineerd op basis van allerlei criteria (belangrijkere processen kunnen bijvoorbeeld ook een prioriteit krijgen) om tot een zo goed mogelijk resultaat te komen.

6.4 System calls & interrupts

Het besturingssysteem staat tussen de processen en de hardware. Een proces spreekt dus niet rechtstreeks de hardware aan. Een paar belangrijke redenen daarvoor zijn:

- De implementatie van de programma's is simpeler, omdat die programma's zich niets moeten aantrekken van de details van de hardware. Het maakt voor een programma niet uit of een bestand is opgeslagen op een SSD of op een harde schijf, of welk bestandssysteem gebruikt wordt (meer over bestandssystemen in hoofdstuk **Error! Reference source not found.**).
- Het besturingssysteem kan controleren wat programma's doen en zo bijvoorbeeld rechten op bestanden afdwingen (zie sectie **Error! Reference source not found.**). Een proces dat draait als gebruiker *bassie* mag bijvoorbeeld niet zomaar de bestanden lezen die enkel leesbaar zijn door de *owner adriaan*.

In deze sectie gaan we bekijken hoe het besturingssysteem op een gecontroleerde manier programma's toegang verleent tot de hardware (*system calls*) en hoe de hardware het besturingssysteem kan op de hoogte houden van bepaalde gebeurtenissen (*interrupts*).



Figuur 6-5: Een schematische voorstelling van *system calls* en *interrupts*

6.4.1 System calls

Een besturingssysteem is eigenlijk zelf een programma dat meer rechten heeft dan eender welk ander proces. Enkel het besturingssysteem heeft rechtstreeks toegang tot de hardware. Als een ander proces iets nodig heeft, laat het dat weten aan de hand van een *system call*. Een *system call* is een beetje te vergelijken met een functieoproep, maar met dat verschil dat het proces de controle doorgeeft aan het besturingssysteem.

Als een proces bijvoorbeeld een `write` system call wil doen om te schrijven naar een bestand, dan zal dat proces pauzeren en zal het besturingssysteem beginnen met uitvoeren. Het besturingssysteem zal nakijken of het proces mag schrijven naar dat bepaald bestand en de hardware aansturen om de gegevens naar het bestand weg te schrijven.

System calls in Linux

Met het commando `strace` kan je achterhalen welke system calls een proces allemaal uitvoert. Als je in de shell voor het commando dat je wil uitvoeren `strace` zet, zal je alle system calls zien die worden uitgevoerd.



Probeer bijvoorbeeld eens `strace ls` te doen. Je zal zien dat dit allerlei system calls uitvoert, zoals `openat` om bestanden te openen, `read` om te lezen, `write` om te schrijven³ en `close` om bestanden weer te sluiten.

Net zoals alle commando's op Linux hebben de system calls ook eigen manpages. Deze bevinden zich allemaal in sectie 2 van de manpages. Met `man 2 write` kan je bijvoorbeeld de manpage van `write` raadplegen. Met `strace` en deze manpages zou je dus kunnen achterhalen wat een programma allemaal doet, zelfs als je de broncode niet hebt.

6.4.2 Interrupts

Als er iets gebeurt, bijvoorbeeld er komt een netwerkpakket toe op de netwerkkkaart, dan moet de hardware dat kunnen laten weten aan het besturingssysteem. Daarvoor wordt een **interrupt** gebruikt.

Als een netwerkpakket toekomt op de netwerkkkaart dan zal de netwerkkkaart een **interrupt request** sturen naar de processor. Dit is een soort bericht dat aan het besturingssysteem laat weten dat het iets moet afhandelen. De processor zal dan het actieve proces onderbreken om naar het besturingssysteem over te gaan.

Wat er gebeurt als er een *interrupt* binnenkomt:

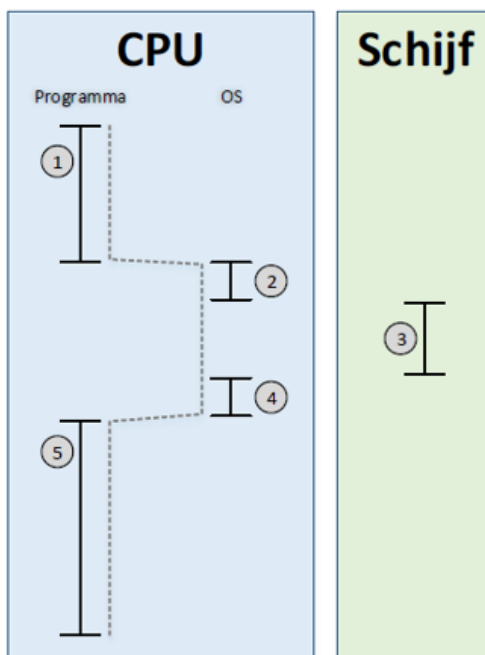
1. De processor stopt het proces in uitvoering.
2. De gegevens (*instruction pointer*, *registers*, ...) over het proces worden opgeslagen in het werkgeheugen.
3. De processor laadt de nodige code (in het besturingssysteem) om de *interrupt* af te handelen.
4. De processor handelt de *interrupt* af.
5. De processor laadt de oorspronkelijke code terug in.

³ `ls` doet één `write` om de output van het commando te schrijven naar de *standard output*. Linux behandelt de *standard output* hetzelfde als een bestand.

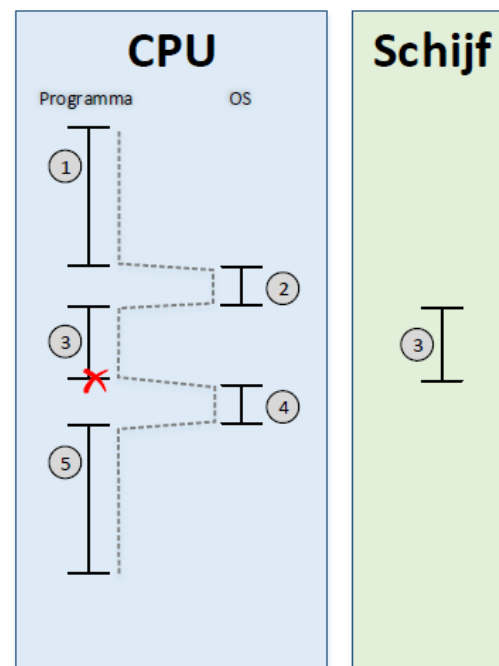
Dit wisselen tussen het proces en uitvoering en een ander proces (in dit geval het besturingssysteem) noemt men een **context switch**.

6.4.3 Snelheidswinst met interrupts

Interrupts worden onder andere ook gebruikt wanneer een programma iets wil lezen van de harde schijf, om te laten weten aan het besturingssysteem dat de leesoperatie gebeurd is. De volgende figuur illustreert dit.



Figuur 6-6: Zonder interrupts

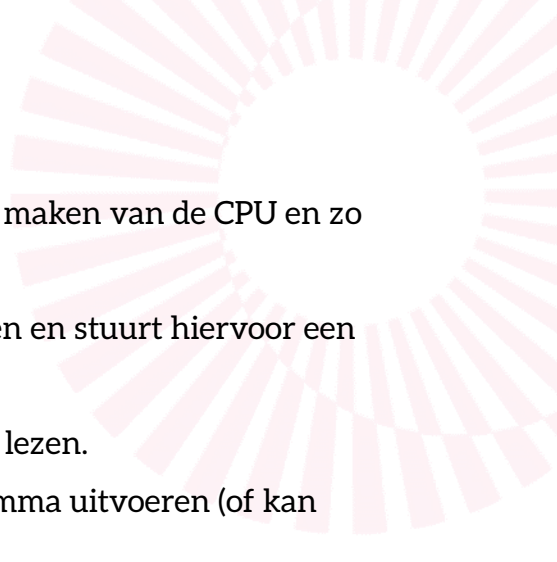


Figuur 6-7: Met interrupts

Zonder *interrupts* (Figuur 6-6) gebeurt het volgende:

1. Het programma wil een bestand van de schijf inlezen en stuurt hiervoor een *system call* naar het OS.
2. Het OS geeft de opdracht aan de schijf om iets uit te lezen.
3. De schijf leest de data terwijl het OS wacht.
4. Het OS kan terug uitvoeren als de data is gelezen.
5. Het proces kan terug hervatten.

We zien dat de CPU geen andere taken kan uitvoeren zolang de leesoperatie bezig is. Dit terwijl er misschien toch andere processen zijn die mogelijk kunnen uitvoeren.



Met interrupts (Figuur 6-7) kunnen we efficiënter gebruik maken van de CPU en zo snelheid winnen:

1. Het programma wil een bestand van de schijf inlezen en stuurt hiervoor een *system call* naar het OS.
2. Het OS geeft de opdracht aan de schijf om iets uit te lezen.
3. Terwijl de schijf de data leest kan een ander programma uitvoeren (of kan het oorspronkelijke proces iets anders doen).
4. Het programma wordt onderbroken door een *interrupt* en het besturingssysteem kan verder het lezen afhandelen.
5. Het oorspronkelijke proces kan terug hervatten.

We zien dat zo de CPU niet in een toestand komt dat die niet benut wordt. Met *interrupts* kunnen we dus efficiënter en bijgevolg sneller werken.