



H3 – Processen

**HO
GENT**

3. Processen

- 3.1 Van programma tot proces
- 3.2 Opbouw van een process
- 3.3 Soorten processen
- 3.4 Beheer van processen
- 3.5 Scheduling

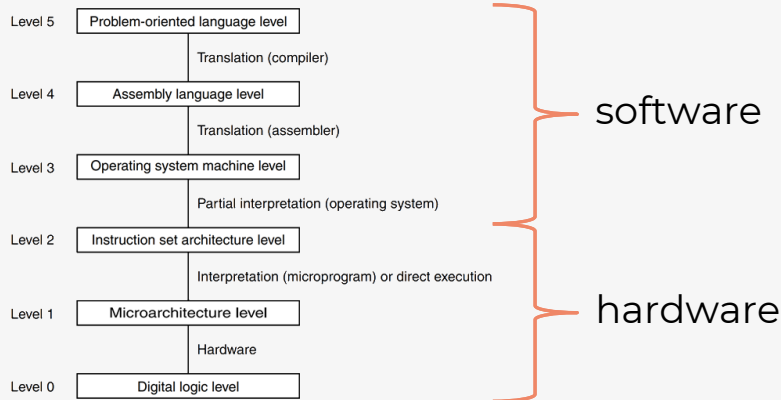
**HO
GENT**

3.1 Van programma tot proces

**HO
GENT**

3.1 Van programma tot proces

Compileren



**HO
GENT**

Wanneer we een programma schrijven in een programmeertaal is dat niet meteen uitvoerbaar door de hardware. De CPU verstaat geen programmeertalen zoals Java, C++, C#, Elke type CPU heeft een bepaalde verzameling instructies die het begrijpt: deze verzameling heet de instructieset van dat type CPU. Elk programma moet dus worden omgezet van programmeertaal tot een verzameling instructies uit die instructieset. Dit proces noemt men “compileren”.

Compileren

i = j + k;	1	ILOAD j	// i = j + k	0x15 0x02
if (i == 3)	2	ILOAD k		0x15 0x03
k = 0;	3	IADD		0x60
else	4	ISTORE i		0x36 0x01
j = j - 1;	5	ILOAD i	// if (i == 3)	0x15 0x01
	6	BIPUSH 3		0x10 0x03
	7	IF_ICMPEQ L1		0x9F 0x00 0x0D
	8	ILOAD j	// j = j - 1	0x15 0x02
	9	BIPUSH 1		0x10 0x01
	10	ISUB		0x64
	11	ISTORE j		0x36 0x02
	12	GOTO L2		0xA7 0x00 0x07
	13 L1:	BIPUSH 0	// k = 0	0x10 0x00
	14	ISTORE k		0x36 0x03
	15 L2:			

(a)

(b)

(c)

Figure 4-14. (a) A Java fragment. (b) The corresponding Java assembly language. (c) The JVM program in hexadecimal.

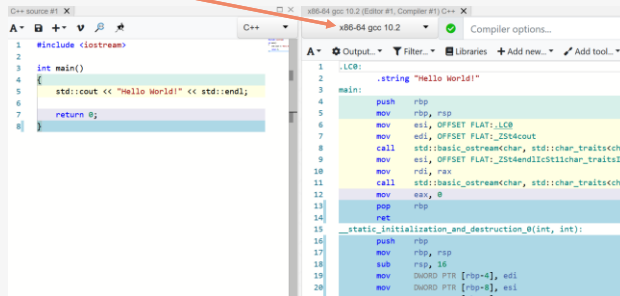
**HO
GENT**

Bovenstaande figuur illustreert hoe een eenvoudig programma in Java omgezet wordt naar een soort van assembler taal. Deze assembler taal wordt dan omgezet naar binaire code (hier voor de eenvoud voorgesteld als hexadecimale code). Merk op dat Java geen gebruik maakt van echte assembler code die rechtstreeks uitgevoerd kan worden door de CPU, maar dat Java gebruikmaakt van Java Assembly Language, een soort van assembleertaal die gebruikt wordt door de Java Virtual machine (zie ook volgende slide).

Instructieset

Uniek per type CPU

Bv. x86, x86-64, ARM, MIPS, JVM, 8051, ...



```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Hello World!" << std::endl;
6 }
7
8 return 0;
9 }
```

```
1 .LC0:
2     .string "Hello World!"
3
4 main:
5     push    rbp
6     mov     rbp, rsp
7     mov     esi, OFFSET FLAT:.LC0
8     mov     edi, OFFSET FLAT:_ZStout
9     call    std::basic_ostreamchar, std::char_traitschar
10    mov     rdi, rax
11    call    std::basic_ostreamchar, std::char_traitschar
12    mov     eax, 0
13    pop     rbp
14    ret
15
16 __static_initialization_and_destruction_0(int, int):
17     push    rbp
18     mov     rbp, rsp
19     sub     rsp, 10
20     mov     QWORD PTR [rbp+8], esi
21     mov     QWORD PTR [rbp+0], edi
22     call    @plt@.L0
```

**HO
GENT**

De instructieset is uniek per type CPU. Er zijn verschillende types:

- x86: De 32-bits instructieset die vroeger door Intel en AMD werd gebruikt voor gebruikers-laptops en –desktops, tegenwoordig is deze vervangen door een uitbreiding hiervan: de 64-bits x86-64 instructieset.
- x86-64: Dit is de instructieset die tegenwoordig door de meeste Intel en AMD processoren gebruikt wordt. Zo goed als alle gebruikers-laptops en –desktops hebben dit type processor.
- ARM: Dit type processor wordt bijvoorbeeld gebruikt op de Raspberry Pi, smartphones, tablets, ... vanwege de lage kost en energieverbruik.
- MIPS: Wordt gebruikt in embedded devices zoals routers, switches, printers, smartphones, tablets, en zelfs supercomputers.
- JVM: De Java Virtual Machine (zie hieronder).
- 8051: Wordt soms gebruikt in embedded devices, daarnaast ook vaak gebruikt voor onderwijs.
- ... : Er zijn uiteraard nog andere CPU types in gebruik of ontwikkeling.

Een programmeur moet dus zijn programma compileren naar elk type CPU waarop het programma moet kunnen uitgevoerd worden. Je ziet bij sommige programma's dat er dus meerdere downloads worden aangeboden.

Een overzicht van de verschillende instructiesets vind je op https://en.wikipedia.org/wiki/Comparison_of_instruction_set_architectures#Instruction_sets.

In de screenshot zie je een C++ programma dat wordt omgezet naar instructies voor het type x86-64. Het programma is een heel eenvoudig programma dat de tekst "Hello World!" toont op de command line en dan afsluit. Op <https://godbolt.org/z/4fe4Yn> kan je in het rechtervenster verschillende types CPU aanduiden en er naar compileren. Je ziet dat de instructies voor elk type telkens verschillen.

Java is hierop een uitzondering (vandaar dat de website werkt met C++). Bij Java wordt er gecompileerd naar bytecodes. Dit is een soort tussentaal met instructies uit een instructieset voor een virtuele machine, de Java Virtual Machine (JVM). De programmeur moet de java code slechts eenmaal compileren naar bytecode, en het programma werkt dan op elk type CPU waarvoor een JVM bestaat. Het is dan immers de taak van de JVM om de bytecode om te zetten naar instructies voor de hardware. Dit bespaart heel wat werk voor de programmeur.

Instructiecyclus

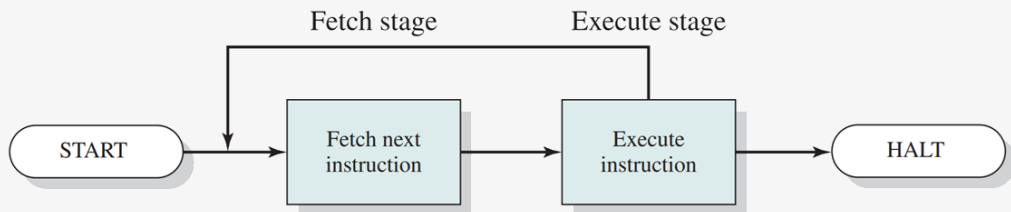


Figure 1.2 Basic Instruction Cycle

**HO
GENT**

De computerhardware werkt steeds oneindig lang volgens hetzelfde stappenplan. Het eenmalig uitvoeren van dit stappenplan wordt ook wel een cyclus genoemd. De hardware voert dus oneindig lang cyclus na cyclus uit. In de meest eenvoudige vorm bestaat zo een cyclus uit 2 stappen, fetch en execute:

- Fetch: Haal de volgende uit te voeren instructie op.
- Execute: Voer deze instructie uit.

Na deze 2 stappen is de cyclus afgewerkt en begint de volgende cyclus.

Instructiecyclus

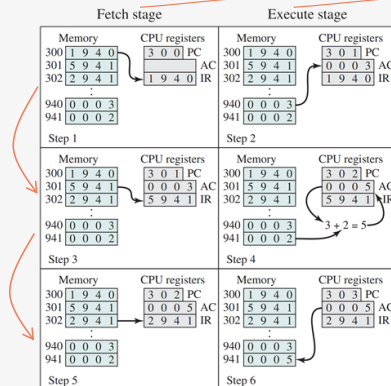


Figure 1.4 Example of Program Execution (contents of memory and registers in hexadecimal)

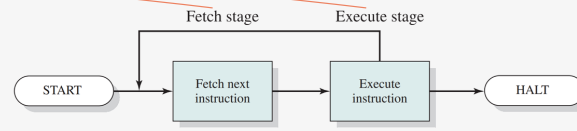


Figure 1.2 Basic Instruction Cycle

**HO
GENT**

In dit voorbeeld wordt een simpel programma uitgevoerd dat 2 getallen optelt en het resultaat opslaat. Dit zou bijvoorbeeld het volgende programma kunnen zijn:

```
int a = 3;
int b = 2;
int c = a + b; // c = 5
```

De instructies voor de hardware van dit systeem zijn als volgt:

- 1XXX: Kopieer de waarde op adres XXX in het geheugen naar het AC register.
- 2XXX: Sla de waarde in het AC register op in het geheugen op adres XXX.
- 5XXX: Tel de waarde op adres XXX in het geheugen op bij de waarde in het AC register en sla het resultaat op in het AC register.

We zien hier de volgende CPU registers:

- PC: Program Counter. Dit register houdt het adres bij van de uit te voeren instructie.
- IR: Instruction Register. Dit register houdt de uit te voeren instructie bij. De hardware analyseert de bits in dit register om de handeling van de instructie uit te voeren.

- AC: ACcumulator. Een hulpregister voor tussenresultaten op te slaan van bewerkingen.

Het programma is gecompileerd naar de volgende instructies van de CPU:

- 1940: Kopieer de waarde op adres 940 in het geheugen naar het AC register
- 5941: Tel de waarde op adres 941 in het geheugen op bij de waarde in het AC register en sla het resultaat op in het AC register
- 2941: Sla de waarde in het AC register op in het geheugen op adres 941

We zien per cyclus telkens de volgende stappen:

1. Fetch: Kopieer de uit te voeren instructie op het adres volgens de PC van het geheugen naar het IR register. Verhoog de PC met 1 zodat deze klaar staat voor de juiste instructie op te halen in de volgende cyclus.
2. Execute: Analyseer de instructie in het IR register en voer deze uit.

Interrupts

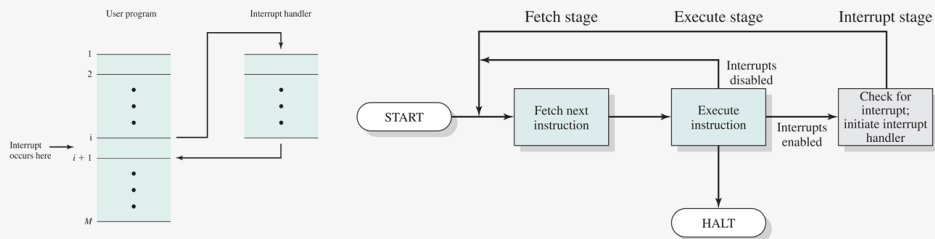


Figure 1.7 Instruction Cycle with Interrupts

**HO
GENT**

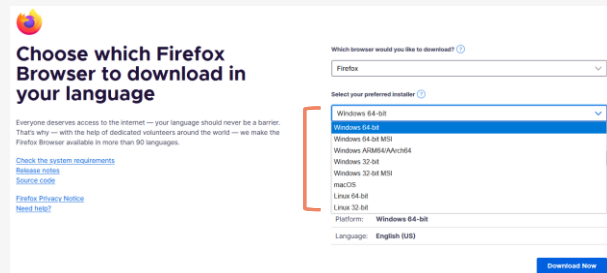
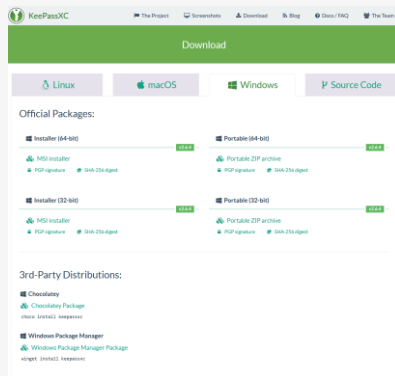
Bij computersystemen treden er soms events op: er wordt een nieuw apparaat aangesloten, de gebruiker voert iets uit, er treed ergens een fout op, er is een timer afgegaan, De computer moet dan hierop kunnen reageren. Zulke onderbreking noemt men een interrupt. De fetch-execute cyclus kan uitgebreid worden om interrupts mogelijk te maken, hiervoor wordt na de execute stap een extra stap voor interrupts toegevoegd in de cyclus. Bij de interrupt stap wordt gekeken of er een interrupt is opgetreden:

- Er is geen interrupt opgetreden: Dan gaat men door naar de volgende cyclus.
- Er is een interrupt opgetreden: De huidige uitvoering van het programma wordt onderbroken. De toestand van het programma wordt opgeslagen. Zo wordt ondermeer het PC register opgeslagen, zodat niet vergeten wordt op welke plek het programma werd onderbroken. Daarna voert het systeem de instructies om de interrupt af te handelen uit via de fetch-execute-cyclus. Het wisselen van het ene programma naar een ander programma wordt ook een context switch genoemd. Deze verzameling instructies noemt men ook wel de interrupt handler. Als de interrupt is afgehandeld, wordt de toestand van het oude programma herstelt en wordt het oude programma verder uitgevoerd.

Het gebruik van context switches komt ook aan bod in het hoofdstuk over scheduling.

3.1 Van programma tot proces

Binaries



**HO
GENT**

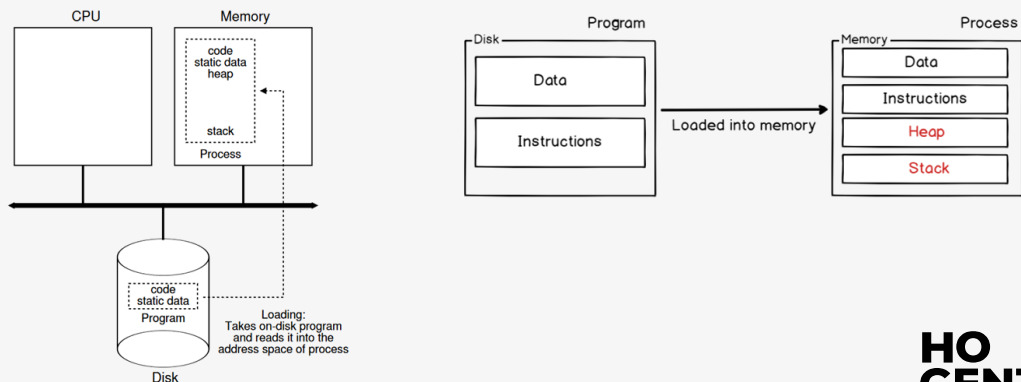
Het bestand gegenereerd door een compiler wordt ook wel een binary of executable genoemd. Op Windows worden programma's gecompileerd naar uitvoerbare bestanden in het PE formaat (Portable Executable). Dit formaat is in de volksmond vaak bekend als het .exe-formaat, alhoewel dit formaat ook wordt gebruikt in bestanden met andere extensies. Bij Linux daarentegen worden programma's gecompileerd naar het ELF format (Executable and Linkable Format). Op Mac wordt dan weer het Mach-O (Mach Object) formaat gebruikt. Een overzicht van dergelijke formaten vind je op

https://en.wikipedia.org/wiki/Comparison_of_executable_file_formats.

Daarnaast bieden verschillende besturingssystemen verschillende functies aan aan programmeurs om bepaalde dingen te doen. Zo is het aanmaken van een venster in een GUI compleet verschillend voor Windows en Linux. Dit zorgt er voor dat een programmeur dus voor elk besturingssysteem een specifieke versie zal moeten maken en compileren. Programma's gecompileerd op het ene besturingssysteem kunnen dus niet zomaar uitgevoerd worden op een ander besturingssysteem. Het is bijvoorbeeld niet mogelijk om een .exe-bestand uit te voeren op Linux. Er bestaan wel programma's die dit proberen mogelijk te maken (zoals WINE op Linux, <https://www.winehq.org/>), maar deze zijn niet altijd bruikbaar.

In de voorbeelden hierboven zie je dat er voor veel programma's verschillende versies beschikbaar zijn voor de meest gebruikte besturingssystemen. Dit is niet bij alle programma's zo: veel games zijn bijvoorbeeld enkel uitvoerbaar op Windows.

Van binary tot proces



**HO
GENT**

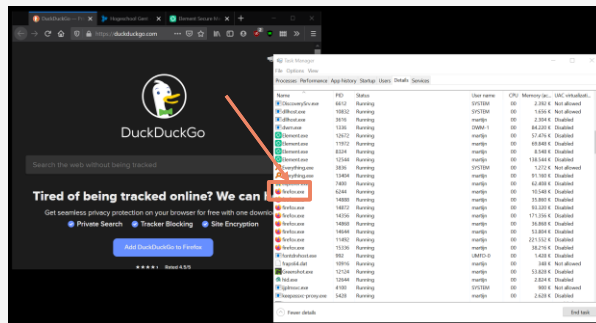
Eenmaal de binary is aangemaakt, is dit simpelweg een bestand dat opgeslagen is op de harde schijf of SSD. Wanneer de gebruiker het programma wil uitvoeren, worden de instructies gekopieerd naar het RAM geheugen. Deze instantie van het programma in het RAM geheugen wordt een proces genoemd. Het bestand op de schijf (het programma) is dus niet hetzelfde als het proces. Het bestand op de schijf (het programma) is iets passief: pas wanneer een instantie daarvan wordt ingeladen in het RAM geheugen krijgen we iets actiefs, namelijk het proces.

3.2 Opbouw van een proces

**HO
GENT**

Proces

Een proces is een **instantie** van een programma dat uitgevoerd wordt op het systeem.



**HO
GENT**

Processen zijn een belangrijk onderdeel van besturingssystemen. Het zijn instanties van programma's die uitgevoerd worden op het systeem. Bijvoorbeeld, stel dat ik een "Hello World!" programma heb gecompileerd naar hello.exe . Als ik dat programma nu 2x activeer door te dubbelklikken, dan draaien er op mijn systeem twee hello.exe programma's, ondanks dat ik slechts één hello.exe bestand heb staan op mijn harde schijf of SSD. Die 2 draaiende programma's zijn instanties van het hello.exe programma en worden elk voorgesteld binnenin het besturingssysteem door processen.

Een proces is dus een soort container die alles bevat om een programma uit te voeren. Dankzij processen kunnen besturingssystemen programma's die worden uitgevoerd beheren. Ook zorgt het gebruik van processen ervoor dat er meerdere programma's tegelijkertijd uitgevoerd kunnen worden.

Een proces in het RAM geheugen kan enkel een bepaald deel van dat RAM geheugen aanspreken om de inhoud van te lezen of er naar te schrijven. Dit gedeelte noemt men de address space van dat proces. De address space bevat onder andere de uitvoerbare instructies van het programma en data van het programma.

Proces Image

- Text: de uit te voeren instructies.
- Data: globale variabelen.
- Stack: tijdelijke opslag voor variabelen, functie parameters, adressen voor return uit functies, ...
- Heap: dynamisch gealloceerd geheugen

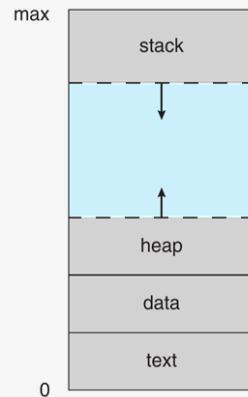


Figure 3.1 Layout of a process in memory.

**HO
GENT**

De process image is hoe het proces er uit ziet in het RAM geheugen. Het is een momentopname van de address space van een proces. Dit bevat in het algemeen de volgende zaken:

- Text: de uit te voeren instructies.
- Data: globale variabelen.
- Stack: tijdelijke opslag voor variabelen, functie parameters, adressen voor return uit functies, Als er geen plaats meer is op deze stack, door bijvoorbeeld een te diepe of oneindige recursie, krijgt men de befaamde stackoverflow error.
- Heap: dynamisch gealloceerd geheugen. Als iets te groot is om op de stack te plaatsen, of als er iets beschikbaar moet zijn over de grenzen van functies heen, dan is de stack geen bruikbare optie. In dat geval kan er gebruik gemaakt worden van de heap om iets dynamisch te alloceren. De heap werkt niet zoals een stack: wat er wordt bewaard, blijft beschikbaar tot het door de programmeur wordt opgekuist.

De exacte vorm van een process image is sterk afhankelijk van besturingssysteem tot besturingssysteem.

Extra informatie (ter info):

- <https://tldp.org/LDP/tlk/kernel/processes.html>
- <https://tldp.org/LDP/LG/issue23/flower/psimage.html>

Proces Control Block

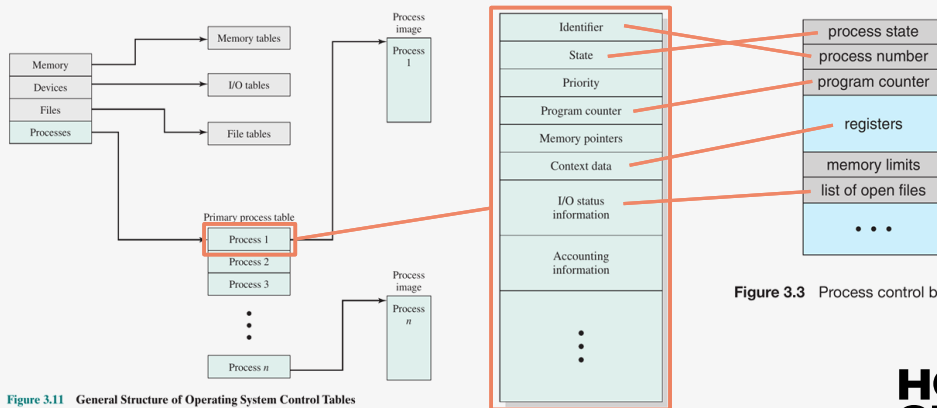


Figure 3.11 General Structure of Operating System Control Tables

Figure 3.1 Simplified Process Control Block

Figure 3.3 Process control block (PCB).

Het besturingssysteem heeft een overzicht van alle processen die actief zijn op het systeem. Dit overzicht heeft de vorm van een tabel en wordt ook wel de process table genoemd. Elke element in de tabel bevat informatie over een proces. Een dergelijk element wordt ook wel een process control block (PCB) genoemd. Een PCB bevat algemeen de volgende informatie:

- Informatie om het proces uniek te identificeren.
- Informatie om de staat van het proces bij te houden bij context switches (zie ook het hoofdstuk over scheduling).
- Informatie om het proces te beheren (zie ook het hoofdstuk over scheduling).

De vorm van een PCB is sterk afhankelijk van besturingssysteem tot besturingssysteem. In de slide tonen we 2 verschillende generieke vormen van een PCB. Je merkt dat bepaalde informatie in beide gevallen terugkomt (al dan niet onder een andere noemer).

3.3 Soorten processen

**HO
GENT**

Soorten processen

- Interactief
- Automatisch
- Daemons

Interactief proces

- Opstarten en controleren vanuit een terminal sessie
- Kunnen zowel in voorgrond (foreground) als achtergrond (background) draaien
- Foreground proces
 - Blokkeert terminal zolang het loopt
- Background proces
 - Blokkeert terminal enkel bij opstart van het proces, nadien kan terminal andere taken uitvoeren

Automatische processen

- Ook wel “batch” processen genaamd
- Verzameling van processen die in een wachtrij worden geplaatst wachtend op uitvoering
- Bij uitvoering worden alle processen één voor één uit de wachtrij uitgevoerd. Dit volgens het **F**irst **I**n, **F**irst **O**ut (FIFO) principe

Voorbeeld: automatisch een back-up maken,
elke dag om middernacht

Daemons

- Processen die continu draaien
- Veelal gestart bij opstarten van systeem
- Wachten in de achtergrond tot ze nodig zijn
- Ook gekend als services

Voorbeeld: onedrive synchronisatie client, e-mail server, ...

3.4 Beheer van processen

**HO
GENT**

Ontstaan

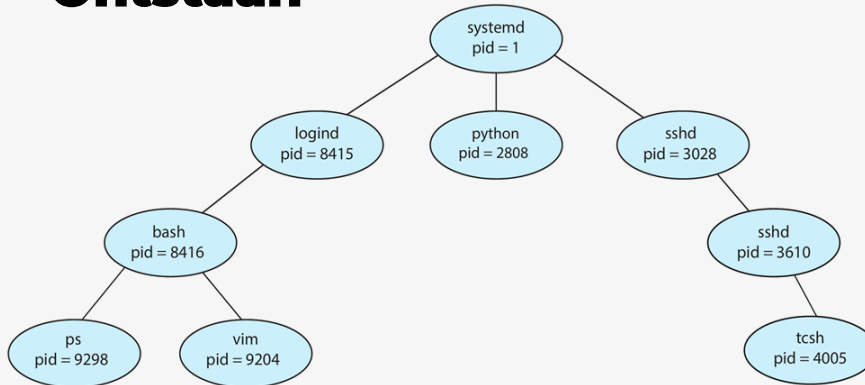


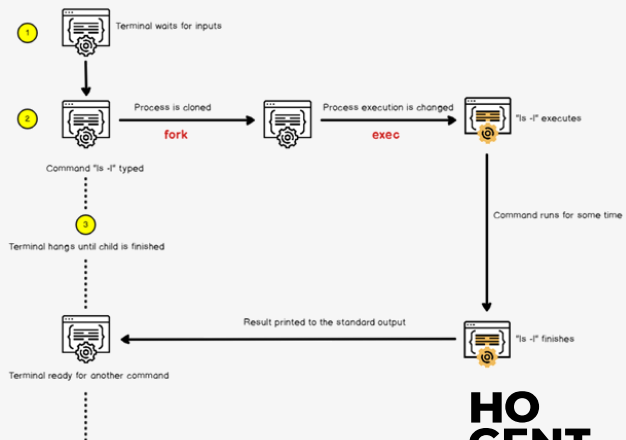
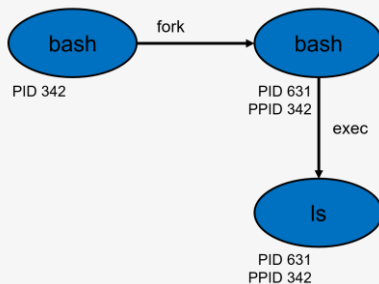
Figure 3.7 A tree of processes on a typical Linux system.

**HO
GENT**

Het ontstaan van processen verschilt van besturingssysteem tot besturingssysteem. We behandelen hier het geval van Linux. Op een linux systeem heeft elk proces een uniek ID-nummer. Dit nummer wordt ook wel Process Identifier (PID) genoemd. Op een linux systeem start bij het opstarten als eerste proces het proces met PID 1. Op recentere systemen is dit systemd, op sommige oudere systemen kan men ook nog het init proces tegenkomen. Systemd is dus bij recente linux distro's de moeder van alle processen. Alle andere processen worden aangemaakt als kinderen door een ouderproces, waardoor er zich op den duur een boom van processen vormt. In het voorbeeld hierboven zien we het moederproces systemd dat de kindprocessen logind (om ervoor te zorgen dat gebruikers zich kunnen inloggen), python (om python programma's uit te voeren), en sshd (om ervoor te zorgen dat gebruikers kunnen inloggen via SSH) heeft aangemaakt. Een gebruiker heeft zich ingelogd op het toestel, waardoor logind het kindproces bashd heeft aangemaakt om de gebruiker een bash shell te geven. Daarin heeft de gebruiker de commando's ps en vim ingetypt en zo bash de opdracht gegeven om de ps en vim kindprocessen aan te maken. Daarnaast zien we ook dat iemand zich heeft ingelogged via SSH (zie het kindproces sshd van sshd) waardoor de tcsh shell is aangemaakt als kindproces van sshd.

Ontstaan

- Fork() and exec()



Het aanmaken van een proces in Linux bestaat uit het aanroepen van 2 functies: `fork()` en `exec()`.

- Fork()**: Deze functie maakt een exacte kopie van het proces in het RAM geheugen (een kopie van diens address space dus) en vult de PID van het proces in met een nieuw ongebruikt procesnummer. Daarnaast vult de functie het PPID (Parent Process Identifier) in met het PID van het ouderproces.
- Exec()**: Het nieuw aangemaakt kindproces wordt overschreven met de nodige waarden voor het gewenste proces. Zo worden bijvoorbeeld de juiste instructies, waarden, ... ingelezen naar het procesbeeld van het kindproces.

Extra informatie:

- <http://cs241.cs.illinois.edu/coursebook/Processes>

3.4 Beheer van processen

Ontstaan

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
```

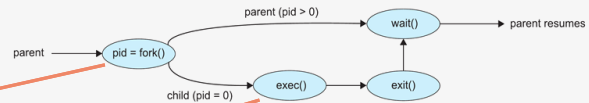


Figure 3.9 Process creation using the fork() system call.

**HO
GENT**

Hier zie je de code geschreven in C++ om een nieuw proces aan te maken. Het is niet belangrijk om de code echt te begrijpen (jullie kennen geen C++), maar wel dat je ziet dat er eerst een `fork()` wordt uitgevoerd en daarna een `exec()` (in dit geval een variant `execlp()`).

Afbraak

- Het proces is afgewerkt of er treedt een fout op:
 - Exit()
- Het proces wordt afgebroken door een ander proces:
 - Bv. Kill
- Soms gaat er iets fout: orphan proces
- In de terminal kan je steeds processen beëindigen met Ctrl+C

**HO
GENT**

Wanneer een proces zijn taak heeft volbracht of er een fout optreedt, wordt het afgesloten. Dit gebeurt in linux door het `exit()` commando. Aan dit commando kan een getal meegegeven worden om aan te duiden dat het proces goed is afgewerkt (getal 0) of dat het is afgesloten door een fout (een ander getal dan 0). Bij een fout vertelt het getal (de foutcode) welke fout er is opgetreden. Als het proces wordt afgesloten, wordt het procesbeeld verwijderd en alle resources (bv. Bestanden, geheugen, ...) dat het proces in gebruik had, worden terug vrijgegeven. De PCB in de process table mag wel nog niet meteen vrijgegeven worden. Het bevat immers de foutcode dat aangeeft waarom het proces is afgesloten. Als de PCB ook meteen verwijderd zou worden, kan het ouderproces niet achterhalen waardoor zijn kindproces is afgesloten. Een proces waarvan enkel de PCB nog bestaat, wordt ook wel een zombieproces genoemd. Normaal gezien bestaan zombieprocessen slechts kortstondig: de PCB wordt snel na het afsluiten van het kindproces gelezen door het ouderproces, waardoor het PCB verwijderd mag worden en het kindproces volledig is verdwenen. Soms gebeurt het helaas dat een foutcode van zo'n zombieproces nooit wordt gelezen (bv. Het ouderproces is afgesloten alvorens de foutcode te lezen). In dat geval wordt het zombieproces een orphan process (weesproces). Het besturingssysteem kan hier niet meer aan en de PCB's van zulke processen vervuilen dus de process table van het besturingssysteem.

2 states

- Not Running
- Running

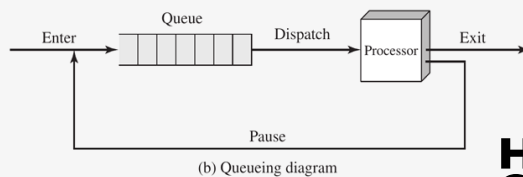
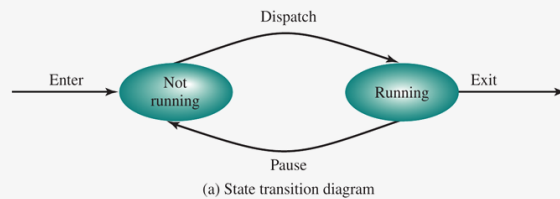


Figure 3.5 Two-State Process Model

**HO
GENT**

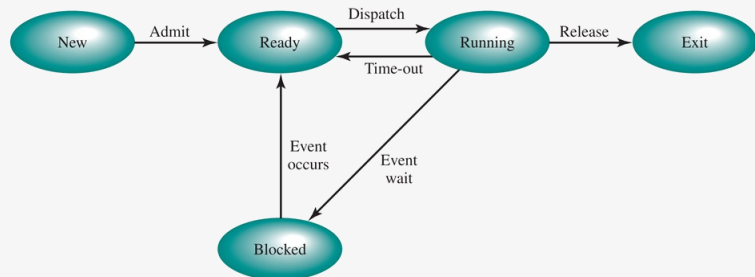
Op dit moment hebben we gezien dat processen zijn eigenlijk in 2 toestanden kunnen bevinden: wordt uitgevoerd op de CPU of staat te wachten. De toestand van een process wordt ook wel state genoemd. Het 2-state scheduling model is het simpelste scheduling model dat er is en bevat dus 2 toestanden:

- Running: dit proces wordt uitgevoerd op de CPU.
- Not Running: dit proces wordt niet uitgevoerd op de CPU.

We zien dat processen die staan te wachten op de CPU in een wachtrij (queue) worden geplaatst en aanschuiven voor CPU tijd. Nieuwe processen worden aangemaakt, krijgen een PCB in de process table, en worden dan toegevoegd aan de queue. Nieuwe processen hebben dus steeds de Not Running toestand (tenzij ze meteen de CPU mogen gebruiken).

5 states

- New
- Ready
- Running
- Blocked
- Exit



**HO
GENT**

Uiteraard zijn de meeste besturingssystemen complexer dan het 2-state scheduling model. Zo kan het ook zijn dat processen staan te wachten op een antwoord van I/O. In dat geval draait het proces niet op de CPU en staat het ook niet in de wachtrij. We hebben dus namelijk een extra toestand nodig: Blocked. We breiden het model uit met de volgende staten:

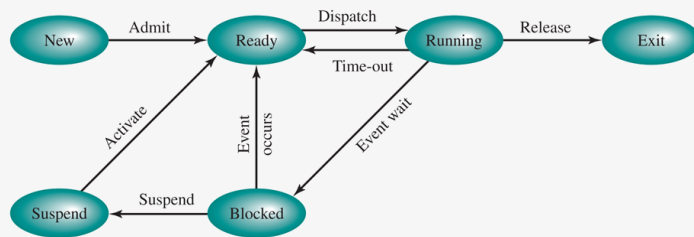
- New: Een nieuw proces aangemaakt door het besturingssysteem. Meestal een nieuw proces waarvan de PCB al is toegevoegd aan de process table, maar dat nog niet volledig is ingeladen in het geheugen. Soms wordt er ook een limiet gezet op het aantal processen in de wachtrij: als deze vol zit mag een proces nog niet overgaan van New naar Ready tot de wachtrij vrije plaatsen heeft.
- Ready: Een proces dat wacht tot het op de CPU mag.
- Running: Het proces wordt uitgevoerd op de CPU.
- Blocked: Een proces dat staat te wachten op iets (zoals lezen/schrijven uit of naar het RAM, geheugen, harde schijf, SSD, netwerk, ...).
- Exit: Een afgewerkt proces. Let op, de exit toestand zegt niets over het feit dat het proces correct is afgewerkt of door een fout is afgesloten. De scheduler heeft daar niets mee te maken, het weet alleen dat hij met dit proces geen rekening moet houden: het is op hoedanook een of andere manier afgerond.

Enkele belangrijke overgangen:

- Ready → Running: Het proces dat als eerste staat in de wachtrij mag op de CPU.
- Running → Ready: Processen mogen vaak slechts een bepaalde maximum tijd op de CPU (zie bv. time sharing). Die maximum tijd wordt ook wel quantum of time slice genoemd. Wanneer de tijd verlopen is, wordt het proces van de CPU gehaald en achteraan de wachtrij geplaatst. Het onderbreken van een proces wordt ook wel pre-emption genoemd.
- Running → Blocked: Het proces moet wachten op iets (bv. I/O) en wordt van de CPU gehaald zodat deze gebruikt kan worden door een ander proces dat de CPU tijd kan gebruiken.
- Blocked → Ready: het proces heeft gedaan met wachten (bv. I/O). Dit is vaak het resultaat van een gebeurtenis (event) (bv. de data is gelezen van de harde schijf). Het proces heeft opnieuw CPU tijd nodig en wordt dus achteraan de wachtrij geplaatst.

6 states

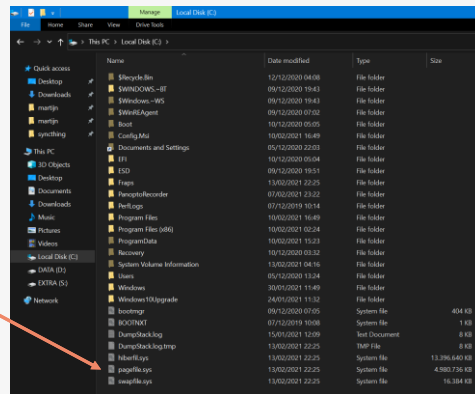
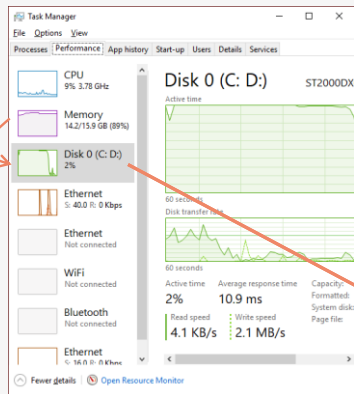
- New
- Ready
- Running
- Blocked
- Suspend
- Exit



**HO
GENT**

Processen in het 5-state scheduling model bevinden zich allemaal in het RAM geheugen van het systeem. Alleen kan het soms gebeuren dat dat RAM geheugen vol geraakt. In dat geval heeft het systeem geen plaats meer om zaken op te slaan of aan te passen en loopt het vast. Om dit op te lossen kan het systeem bepaalde processen die staan de wachten uit het RAM geheugen verplaatsen naar de harde schijf of SSD om RAM geheugen vrij te maken. Dit wordt ook wel swapping genoemd. Zulke processen bevinden zich dan in de Suspend toestand. Wanneer het systeem terug voldoende ademruimte heeft in het RAM geheugen, kan het systeem terug processen vanop de hardeschijs of SSD verplaatsen naar het RAM geheugen. De processen krijgen dan terug de Ready toestand en mogen weer aanschuiven in de wachtrij.

Swapping (suspend)



**HO
GENT**

Op dit systeem zie je dat het RAM geheugen volloopt. Het systeem riskeert daardoor vast te lopen. Om dat te vermijden zal het systeem processen wegschrijven naar de harde schijf of SSD om RAM geheugen vrij te maken om ervoor te zorgen dat het systeem niet in de problemen komt. Op Windows wordt er weggeschreven naar de pagefile.sys en swapfile.sys bestanden op de C:\ schijf (meer info over deze bestanden vind je op <https://helpdeskgeek.com/help-desk/hdg-explains-swapfile-sys-hiberfil-sys-and-pagefile-sys-in-windows-8/>). Op Linux is er daarentegen vaak een aparte swap partitie aanwezig.

Swappen is een intensief proces: de harde schijf of SSD is vele malen trager dan het RAM geheugen (zie de tabel op een aantal slides terug). Swappen vertraagt het systeem enorm!

Er is tegenwoordig ook een debat of swap ruimte of Windows of Linux met de huidige RAM hoeveelheden nog relevant is. Het heeft zijn voor- en nadelen om wel of geen swap ruimte te voorzien. Meer info vind je op:

- <https://www.windowscentral.com/what-swapfilesys-and-do-i-need-it-my-windows-10-pc>
- <https://www.redhat.com/en/blog/do-we-really-need-swap-modern-systems>

3.5 Scheduling

**HO
GENT**

Multiprogramming

- CPU zo optimaal mogelijk gebruiken
 - I/O is trager dan CPU
 - Sommige processen moeten wachten op iets (zoals bv. I/O)
- Wanneer een proces moet wachten, kan een ander proces gebruik maken van de CPU

$$CPU\ Utilization = \frac{CPU\ Usage\ Time}{Total\ Time}$$



Streven naar 100 %

Best met een marge: zie notes (*)

$$CPU\ Idle = \frac{CPU\ Idle\ Time}{Total\ Time}$$



Streven naar 0 %

**HO
GENT**

Een besturingssysteem zal steeds trachten de hardware zo optimaal mogelijk te benutten. Aangezien CPU tijd een schaars goed is, zal het de CPU ten volle proberen gebruiken. Het percentage tijd dat de CPU in gebruik is, wordt ook wel CPU utilization genoemd. Als een CPU staat te nietsen, zegt men dat de CPU idle is. Het besturingssysteem probeert de CPU utilization zo hoog mogelijk te krijgen en de CPU idle time zo laag mogelijk. Hierbij zijn wel enkele valkuilen. Zo is de snelheid van een CPU vele malen hoger dan die van andere hardware zoals RAM, harde schijven, SSD's, netwerk, Elke keer een programma iets van de opslag of netwerk nodig heeft, moet het wachten. Hierdoor gaat kostbare tijd verloren. Om de CPU toch bezig te houden wordt er ondertussen een ander proces op de CPU geplaatst. Wanneer het eerste proces gedaan heeft met wachten op de I/O, vraagt het terug CPU tijd aan. Dit concept wordt multiprogramming genoemd en zorgt ervoor dat de CPU continue bezig blijft en er geen CPU tijd verloren gaat.

(*) Een besturingssysteem dat continue aan 100% draait is ook niet ideaal. Indien er dan een interrupt binnenkomt, is er immers geen CPU tijd beschikbaar om deze interrupt af te werken. M.a.w. een systeem waarvan de CPU volledig aan 100% werkt, zal niet meer lijken te reageren tot de taken die de CPU opeisen afgewerkt zijn of minder CPU tijd vragen. Er wordt dus best gestreefd naar een goed compromis tussen

een zo hoog mogelijke CPU utilization en een marge voor interrupts. De ideale CPU utilization is dus eigenlijk minder dan 100%.

Multiprogramming

Computer Action	Avg Latency	Normalized Human Time
3GhzCPU Clock cycle 3Ghz	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
RAM access	70 - 100ns	3.5 to 5.5 min
NVMe SSD I/O	7-150 μ s	2 hrs to 2 days
Rotational disk I/O	1-10 ms	11 days to 4 mos
Internet: SF to NYC	40 ms	1.2 years
Internet: SF to Australia	183 ms	6 years
OS virtualization reboot	4 s	127 years
Virtualization reboot	40 s	1200 years
Physical system reboot	90 s	3 Millenia

Table 1: Computer Time in Human Terms¹

**HO
GENT**

Bron: <https://formulusblack.com/blog/compute-performance-distance-of-data-as-a-measure-of-latency/>

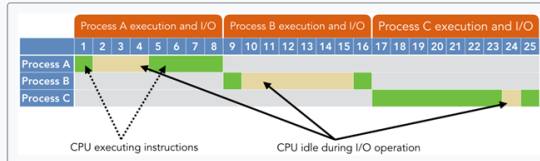
3.5 Scheduling

Multiprogramming



Example 2.2.1

Consider the following timeline illustration for three sequential uniprogramming processes.



The green regions indicate times when the CPU is executing instructions in the program, while the yellow indicates that the times where the CPU is idle while waiting on an I/O operation to complete. The following table summarizes the time each process spends executing on the CPU or waiting for I/O:

Process	CPU time	I/O time
A	5	3
B	2	6
C	8	1
Total	15	10

In this scenario, the CPU was used for a total of 15 out of 25 possible seconds, so the system experienced 60% CPU utilization when running these three jobs:

$$\text{CPU utilization} = \frac{5 + 2 + 8}{5 + 2 + 8 + 3 + 6 + 1} = \frac{15}{25} = 60\%$$

**HO
GENT**

Bron:

<https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/Multiprogramming.html>

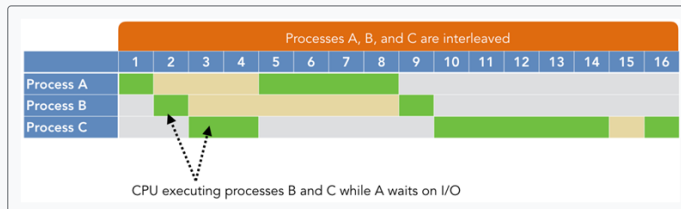
3.5 Scheduling

Multiprogramming



Example 2.2.2

Consider the following timeline illustration for the same three processes from [Example 2.2.1](#), but in a multiprogramming environment.



As before, the green regions indicate CPU execution and the yellow indicates I/O operations. However, note that processes B and C can run while A is waiting on its I/O operation. Similarly, A and C execute while B is waiting on I/O operations. As a result, the CPU is only completely idle while C's I/O operation is performed at time 15, because A and B have already run to completion.

In our revised CPU utilization calculation, the numerator does not change because the total amount of CPU execution time has not changed. Only the denominator changes, to account for the reduced time wasted waiting on A's and B's I/O operations.

$$\text{CPU utilization} = \frac{5 + 2 + 8}{5 + 2 + 8 + 1} = \frac{15}{16} = 93.75\%$$

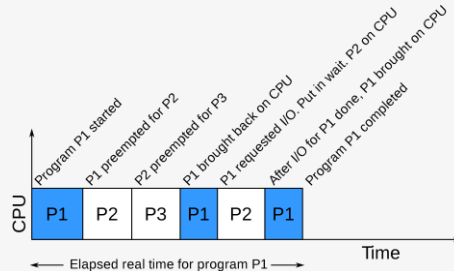
**HO
GENT**

Bron:

<https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/Multiprogramming.html>

Time sharing

- Multitasking
- Multi-user
- Processen op de CPU wisselen elkaar (snel) af
- Geeft de illusie dat processen tegelijkertijd worden uitgevoerd



**HO
GENT**

Terwijl multiprogramming dient om geen CPU tijd verloren laten gaan, gebruikt het besturingssysteem ook processen om de illusie te geven dat de gebruiker meerdere processen tegelijkertijd uitvoert. In de realiteit wisselt het besturingssysteem processen op de CPU elkaar kort af, waardoor het lijkt alsof de processen tegelijkertijd uitgevoerd worden. Multiprogramming kan dus ook perfect werken op een CPU met slechts 1 core. Het besturingssysteem moet wel een goede afweging maken hoelang processen aan een stuk op de CPU mogen. Als deze tijd te kort is, dan steekt het besturingssysteem te veel tijd in het wisselen van processen en is het niet efficiënt. Is de tijd te lang, dan merkt de gebruiker dit op.

Multiprogramming en time sharing hebben dus een verschillend doel:

- Multiprogramming: de CPU ten volle benutten.
- Time sharing: de illusie van parallele processen creëren.

Context switch

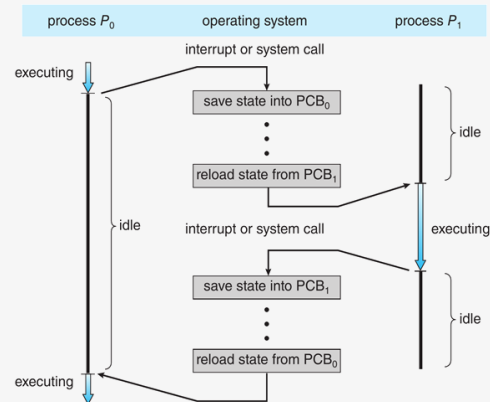


Figure 3.6 Diagram showing context switch from process to process.

**HO
GENT**

Bij het wisselen van een proces treedt een context switch op: er wordt een snapshot genomen van het volledige proces en bewaard in het geheugen. Dan wordt de snapshot van het andere proces ingeladen en dat proces uitgevoerd. Als het terug de beurt is aan het eerste proces, wordt er opnieuw een snapshot van het tweede proces bewaard en de snapshot van het eerste proces ingeladen. Het proces doet verder vanwaar de snapshot is opgenomen alsof er niets is gebeurd. Het nemen van een snapshot heeft wel een kleine overhead: verschillende data moeten worden opgeslagen, zoals de stack, heap, data, registers (PC, ...), ... en dit neemt een klein beetje tijd in beslag. Hetzelfde geldt voor het terug inladen van een snapshot.

Scheduler

- Verantwoordelijk voor het bepalen wanneer welk proces CPU tijd krijgt
- Afweging tussen verschillende soorten processen:
 - Batch
 - Interactive
 - Real time
- Hoe deze beslissing nemen? → Niet eenvoudig

**HO
GENT**

De scheduler is het onderdeel van het besturingssysteem dat beslist wanneer een bepaald proces CPU tijd krijgt. De simpelste scheduler is een eenvoudige wachtrij: het eerste proces in de wachtrij mag uitgevoerd worden. Wanneer dat klaar is, is het aan het volgende proces. Nieuwe processen moeten achteraan de wachtrij (queue in het Engels) aanschrijven tot het hun beurt is. Dit eenvoudig algoritme noemt men First Come First Serve. Er zijn nog veel complexere algoritmes mogelijk waarbij er gekeken wordt naar hoeveel een proces moet wachten op I/O, hoelang een proces zal duren, welke processen prioriteiten krijgt,

Een scheduler heeft dus een lastige taak en is ook vaak een lastige programmeeroefening. Wanneer een scheduler zijn werk niet goed doet, heeft dit vaak gevolgen voor de gebruiker. We delen de mogelijke types processen in in 3 groepen:

- Batch processen zijn processen die een na een moeten worden uitgevoerd en vaak weinig tot geen interactie hebben met de gebruiker. Vaak gaat het om een lijst van opdrachten die gegeven wordt aan de computer, die het systeem een na een afwerkt. Bij batchprocessen is een wachtrij systeem vaak voldoende.
- Interactieve processen zijn de processen op command line of GUI waarmee we het meeste vertrouwt zijn. We kunnen deze uitvoeren door zaken in te typen of door

op knoppen te duwen. Deze processen geven ook vaak resultaten terug weer. Als de scheduler niet goed zijn job doet lijkt het voor de gebruiker alsof programma's blijven hangen.

- Real time systemen zijn systemen die een hoge snelheidsrespons nodig hebben. Bij streamen bijvoorbeeld leiden kleine vertragingen al snel tot stotter en lag, wat erg hinderlijk is voor de gebruiker. Schedulers doen er dus goed aan om zulke processen voorrang te geven.

Het ontwikkelen en programmeren van een scheduler is dus vaak een complexe uitdaging.

Extra bronnen over queues:

- https://en.wikibooks.org/wiki/A-level_Computing/AQA/Paper_1/Fundamentals_of_data_structures/Queues
- https://everythingcomputerscience.com/discrete_mathematics/Stacks_and_Queues.html

Preemptive

- = Onderbreken huidig proces
- Nodig indien proces systeem kan monopoliseren
- Wisselen van proces zorgt voor extra overhead

Starvation

- In sommige gevallen kan het zijn dat bepaalde processen nooit CPU tijd krijgen
- Mogelijk scenario:
 - De scheduler geeft voorrang aan korte processen
 - Er komen steeds nieuwe korte processen in het systeem
 - Hierdoor worden lange processen telkens uitgesteld

Voorbeelden van type schedulers

- Zonder onderbrekingen
 - First Come, First Serve (FCFS)
 - Shortest Process Next (SPN)
- Met onderbrekingen
 - Shortest Remaining Time (SRT)
 - Round Robin (RR)

**HO
GENT**

Er bestaan verschillende algoritmen voor schedulers om CPU tijd toe te wijzen aan processen. Deze kunnen onderverdeeld worden in 2 categorieën:

- Non-preemptive: algoritmes die geen gebruik maken van context switches en dus pas wisselen van proces wanneer een proces volledig is afgewerkt.
- Preemptive: algoritmes die gebruik maken van context switches om processen op de CPU te onderbreken en om te wisselen met een ander proces.

In de volgende slides zien we van elk type een voorbeeld van een eenvoudig algoritme:

- FCFS (First Come First Serve): een non-preemptive algoritme waarbij er gebruik wordt gemaakt van een eenvoudige wachtrij queue. Elk proces op de CPU wordt volledig afgewerkt. Daarna is het de beurt aan het volgende proces uit de wachtrij, dan dat dan op zijn beurt ook volledig wordt afgewerkt. Er worden geen processen van de CPU gehaald als ze nog niet volledig zijn afgewerkt.
- SPN (Shortest Proces Next): een non-preemptive algoritme waarbij er gebruik wordt gemaakt van een eenvoudige wachtrij queue. Elk proces op de CPU wordt volledig afgewerkt. Daarna is het de beurt aan het volgende proces dat de kortste

uitvoeringstijd nog heeft uit de wachtrij, dan dat dan op zijn beurt ook volledig wordt afgewerkt. Er worden geen processen van de CPU gehaald als ze nog niet volledig zijn afgewerkt.

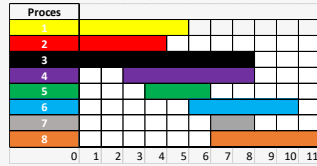
- SRT (Shortest Remaining Time): een preemptive algoritme waarbij er telkens een nieuw proces ontstaat een afweging wordt gemaakt welk proces het minst tijd op de CPU zal nemen. Als het nieuw proces minder CPU tijd nodig heeft dan het huidige proces op de CPU, wordt er van proces gewisseld. Er wordt dus gebruik gemaakt van context switches.
- RR (Round Robin): een preemptive algoritme waarbij de CPU om de Q eenheden wisselt naar het volgende proces uit de wachtrij. Het huidige proces wordt dan achteraan in de wachtrij opnieuw toegevoegd indien het nog niet klaar was.

First Come, First Served (FCFS)

- Proces wordt volledig uitgevoerd, geen onderbrekingen
 - Processen worden in wachtrij geplaatst indien nodig
 - Volgende proces? Eerst proces uit de wachtrij
- + : Geen starvation, heel gemakkelijk
- : Korte processen kunnen lang moeten wachten, Proces kan systeem monopoliseren

3.5 Scheduling

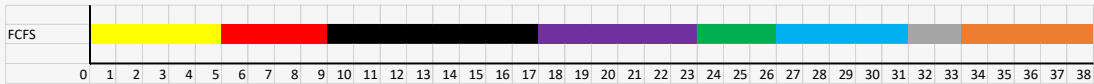
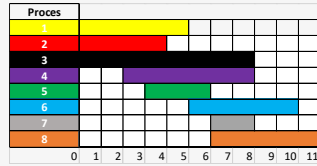
FCFS



- Tabel bevat overzicht van alle processen
- Horizontale as = tijdslijn
 - Startpunt = tijdstip dat proces aangeboden wordt
 - Aantal gekleurde vakjes = tijd dat proces nodig heeft om uitgevoerd te worden
 - B.v.: groene proces
 - Start op tijdstip 3
 - 3 tijdseenheden nodig om uitgevoerd te worden

3.5 Scheduling

FCFS

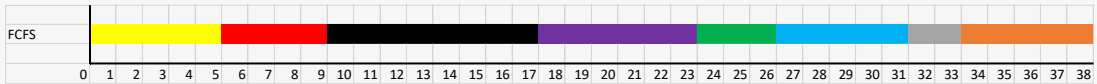
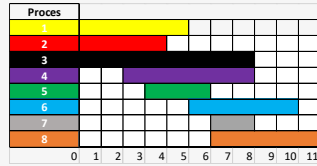


- Bovenstaande tabel bevat overzicht van alle processen na planning volgens FCFS methode volgens tijdslijn
- B.v.: groene proces
 - Effectief gestart na 23 tijdseenheden
 - Volledig afgerond na 26 tijdseenheden

**HO
GENT**

3.5 Scheduling

FCFS



Tabel rechts bevat voor de planner volgende informatie

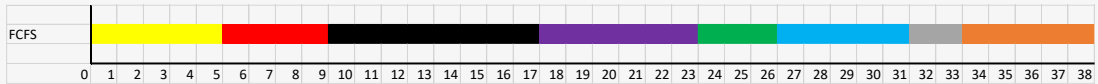
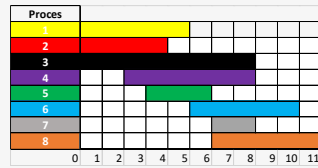
- **Wachttijd:** tijd dat het proces in het systeem aanwezig was maar niet uitgevoerd werd
- **Turnaround time:** tijd dat het proces in het systeem aanwezig was vanaf aanbieden tot effectief uitgevoerd zijn (= wachttijd + uitvoeringstijd)

Proces	FCFS	
	Wachttijd	Turnaround time
1	0	5
2	5	9
3	9	17
4	15	21
5	20	23
6	21	26
7	25	27
8	27	32
Totaal	122	160
Gemiddeld	15,25	20

**HO
GENT**

3.5 Scheduling

FCFS



Proces	FCFS	
	Wachttijd	Turnaround time
1	0	5
2	5	9
3	9	17
4	15	21
5	20	23
6	21	26
7	25	27
8	27	32
Totaal	122	160
Gemiddeld	15,25	20

**HO
GENT**

Shortest Process Next (SPN)

- Proces wordt volledig uitgevoerd, geen onderbrekingen
- Processen worden in wachtrij geplaatst indien nodig
- Volgende proces? Kortste proces uit de wachtrij

+ : Korte processen zijn snel uitgevoerd

- : Starvation voor lange processen, Proces kan systeem monopoliseren

**HO
GENT**

SPN

HO GENT

Shortest Remaining Time (SRT)

- Proces wordt onderbroken, indien er zich een beter (= korter) proces aandient
- Processen worden in wachtrij geplaatst indien nodig
- Volgende proces? Proces met kortste overgebleven uitvoeringstijd uit de wachtrij

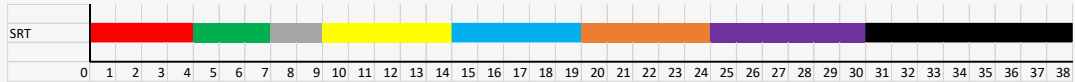
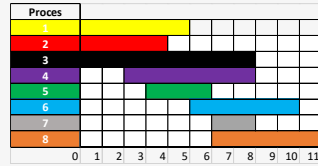
+ : Korte processen zijn snel uitgevoerd

- : Starvation voor lange processen, Overhead bij vele wisselen

**HO
GENT**

3.5 Scheduling

SRT



Proces	SRT	
	Wachttijd	Turnaround time
1	9	14
2	0	4
3	30	38
4	22	28
5	1	4
6	9	146
7	1	3
8	13	18
Totaal	85	123
Gemiddeld	10,625	15,375

**HO
GENT**

- Toeval hier dat SRT = SPN. Is uiteraard niet altijd zo!

Round Robin (RR)

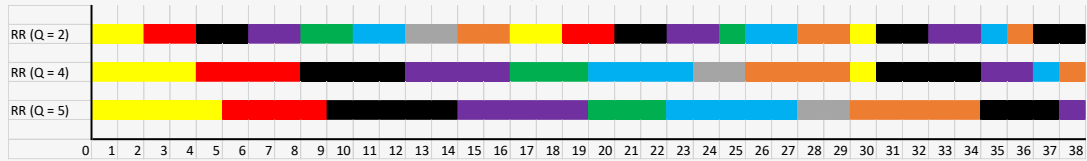
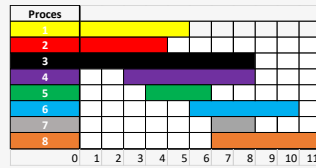
- Elk proces wordt beurtelings Q eenheden uitgevoerd
- Processen worden in wachtrij geplaatst indien nodig
- Volgende proces? Volgende wachtrij uit wachtrij, laatst uitgevoerde wordt achteraan terug toegevoegd

+ : Eerlijk, geen starvation

- : Waarde voor Q heel belangrijk, Korte processen moeten soms lang wachten, Overhead bij vele wisselen

3.5 Scheduling

RR

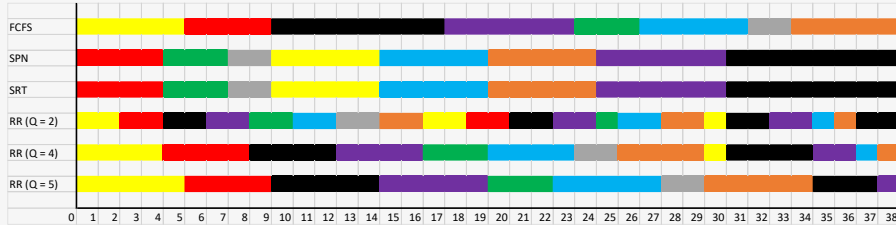
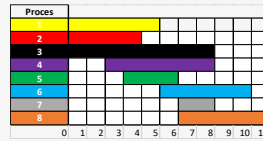


Proces	RR (Q = 2)		RR (Q = 4)		RR (Q = 5)	
	Wachttijd	Turnaround time	Wachttijd	Turnaround time	Wachttijd	Turnaround time
1	25	30	0	5	0	5
2	16	20	4	8	5	9
3	30	38	26	34	29	37
4	26	32	28	34	30	36
5	19	22	13	16	16	19
6	25	30	27	32	17	22
7	6	8	17	19	21	23
8	25	30	27	32	23	28
Totaal	172	210	142	180	141	179
Gemiddeld	21,5	26,25	17,75	22,5	17,625	22,375

HO
GENT

3.5 Scheduling

Overzicht



	FCFS		SPN		SRT		RR (Q = 2)		RR (Q = 4)		RR (Q = 5)	
Proces	Wachttijd	Turnaround time	Wachttijd	Turnaround time	Wachttijd	Turnaround time	Wachttijd	Turnaround time	Wachttijd	Turnaround time	Wachttijd	Turnaround time
1	0	5	9	14	9	14	25	30	0	5	0	5
2	5	9	0	4	0	4	16	20	4	8	5	9
3	9	17	30	38	30	38	30	38	26	34	29	37
4	15	21	22	28	22	28	26	32	28	34	30	36
5	20	23	1	4	1	4	19	22	13	16	16	19
6	21	26	9	14	9	14	25	30	27	32	17	22
7	25	27	1	3	1	3	6	8	17	19	21	23
8	27	32	13	18	13	18	25	30	27	32	23	28
Totaal	122	160	85	123	85	123	172	210	142	180	141	179
Gemiddeld	15,25	20	10,625	15,375	10,625	15,375	21,5	26,25	17,75	22,5	17,625	22,375

**HO
GENT**

- Beste planner in deze situatie was SPN of SRT

Overzicht type planners

	FCFS	SPN	SRT	RR
Preemptive	Nee	Nee	Ja	Ja
Voordelen	<ul style="list-style-type: none"> • Geen starvation • Gemakkelijk 	Korte processen snel uitgevoerd	Korte processen snel uitgevoerd	<ul style="list-style-type: none"> • Geen starvation • Eerlijk
Nadelen	Korte processen moeten soms lang wachten	<ul style="list-style-type: none"> • Starvation lange processen • Monopolisatie systeem mogelijk 	<ul style="list-style-type: none"> • Starvation lange processen • Overhead bij vele wisselen 	<ul style="list-style-type: none"> • Q waarde belangrijk • Korte processen moeten soms lang wachten
Beste bij	Lange processen	Korte processen	Korte processen	Lange processen

**HO
GENT**

- Beste planner in deze situatie was SPN of SRT

**HO
GENT**

Bronnen

- Andrew S. Tanenbaum and Herbert Bos. 2014. *Modern Operating Systems* (4th. ed.). Prentice Hall Press, USA
- William Stallings. 2018. *Operating Systems: Internals and Design Principles, 9/e* (9th. ed.). Pearson IT Certification, Indianapolis, Indiana, USA
- Tanenbaum & Austin, 2013. *Structured Computer Organization* (6th. ed.). Pearson, USA
- Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. 2018. *Operating System Concepts* (10th. ed.). Wiley Publishing
- Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2018. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books. <https://pages.cs.wisc.edu/~remzi/OSTEP/#book-chapters>

**HO
GENT**