# Les 2: Processen en draden

The speedup of a parallel program is limited by the time needed for the sequential fraction of the program -- Amdahl's law (Gene Amdahl)

Any sufficiently large problem can be efficiently parallelized -- Gustafson's Law (John Gustafson)

best2-1

# Overzicht

- Processen
- Draden
- Voorbeelden

best2-2

# Proces

- Proces = een **programma in uitvoering**, inclusief alle toestand: geheugen, registers, code, programmateller, …
- Processen delen geen systeemmiddelen!
- Eén core voert op elk moment ten hoogste 1 proces uit.
- Naast gebruikersprocessen ook systeemprocessen

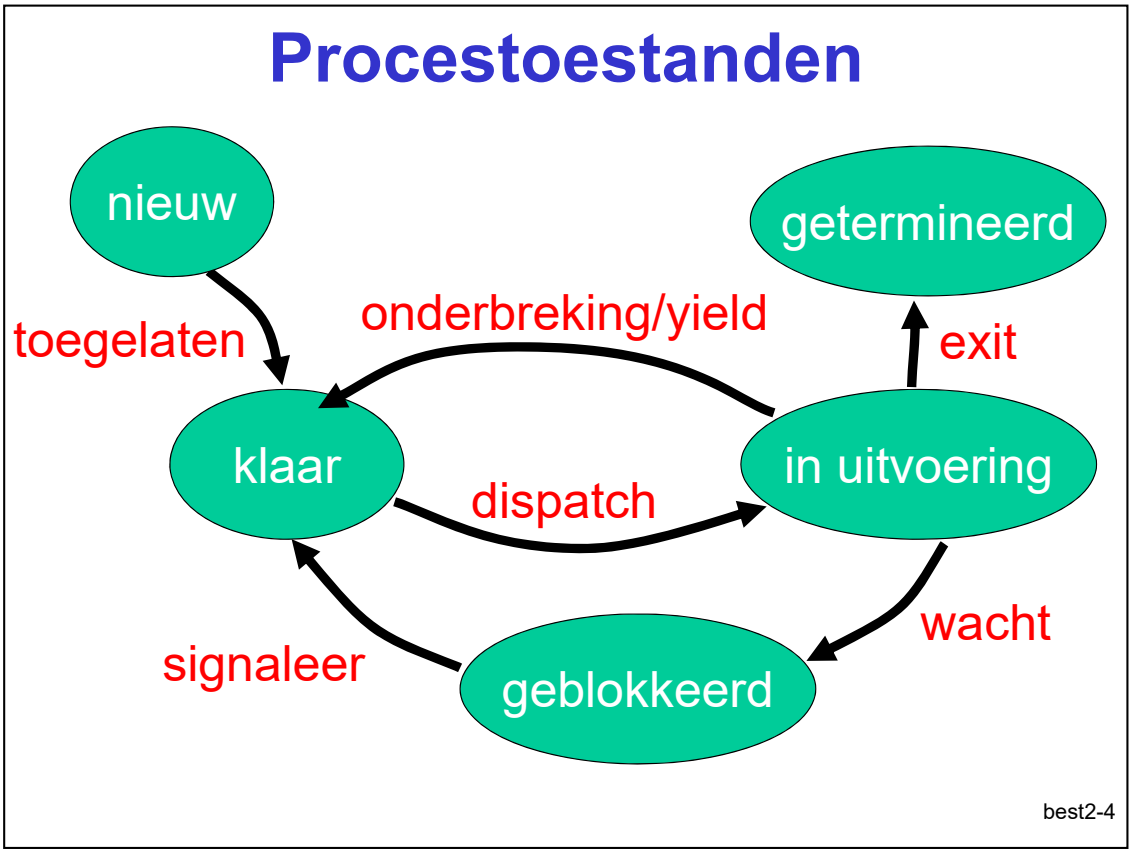
best2-3

Een proces is een activatie van een programma. Het bestaat uit alles wat er nodig is om een programma te kunnen uitvoeren: instructies, grabbelgeheugen, stapel, registers, programmateller, bestanden, randapparaten, … Men maakt een onderscheid tussen gebruikersprocessen (opgestart door de gebruiker) en systeemprocessen (opgestart door het systeem, vaak met extra privileges).

Processen delen in principe geen systeemmiddelen en draaien dus compleet onafhankelijk van elkaar. Dit is vereist om er te kunnen voor zorgen de processen elkaar niet kunnen schaden.

Indien het finaal toch nodig blijkt dat processen met elkaar moeten interageren, dan dient men dit expliciet duidelijk te maken door gebruik te maken van interprocescommunicatieprimitieven zoals boodschappen om uit te wisselen, of een stuk geheugen dat gedeclareerd wordt als zijnde gemeenschappelijk tussen de verschillende processen.

In de regel kan er op elk moment per processor slechts 1 proces actief zijn. Op een multiprocessorsysteem kunnen er uiteraard evenveel processen actief zijn als er processors in het systeem aanwezig zijn.




Een proces in uitvoering kan zich in verschillende toestanden bevinden. Meteen na de creatie bevindt het zich in de toestand nieuw. Op interactieve systemen wordt een gecreëerd proces meteen ook toegelaten tot de toestand klaar. Op grote batchsystemen beslist de jobplanner of een proces al dan niet tot de klaartoestand toegelaten wordt. Een reden om dat niet te doen kan zijn dat de machine al 100% belast is met andere jobs en dat het toelaten van een extra job er niet zal voor zorgen dat de machine efficiënter gebruikt wordt. Bij interactieve systemen moeten alle processen na hun creatie uiteraard toegelaten worden.

De processen in de toestand klaar staan klaar om uitgevoerd te worden, maar worden nog niet uitgevoerd. Ze staan te wachten op het beschikbaar komen van de processor. Eenmaal gekozen gaat een dergelijk proces over naar de toestand in uitvoering. Op een monoprocessor kan er zich slechts één proces in deze toestand bevinden; op een multiprocessor evenveel als er beschikbare processors zijn. Vanuit de toestand in uitvoering kan het proces terugkeren naar de toestand klaar (indien zijn tijdskwantum voor de uitvoering opgebruikt is, of indien het proces spontaan de processor afstaat door de oproep yield uit te voeren), of het proces kan overgaan naar de toestand getermineerd (door het uitvoeren van exit), of het proces kan terechtkomen in de toestand geblokkeerd. Dit laatste gebeurt indien er een systeemoproep gebeurt die niet meteen kan beantwoord worden (b.v. het afhalen van het volgende teken van het toetsenbord).

Vanuit de toestand geblokkeerd zal het proces terug naar de klaar toestand verhuizen van zodra de oorzaak van het wachten opgeheven werd (er werd b.v. een toets ingedrukt). Het feit dat een proces niet langer geblokkeerd is, betekent niet dat het proces zal gaan uitvoeren. Dit zal het pas doen nadat het proces geselecteerd werd voor uitvoering.

Een proces kan zich een tijdlang in de getermineerde toestand bevinden. Dit is nodig totdat duidelijk wordt dat geen enkel proces nog geïnteresseerd zal zijn in het resultaat van de berekening van dit proces (b.v. van zodra dat het ouderproces termineert).

# Procescontroleblok (PCB)


best2-5

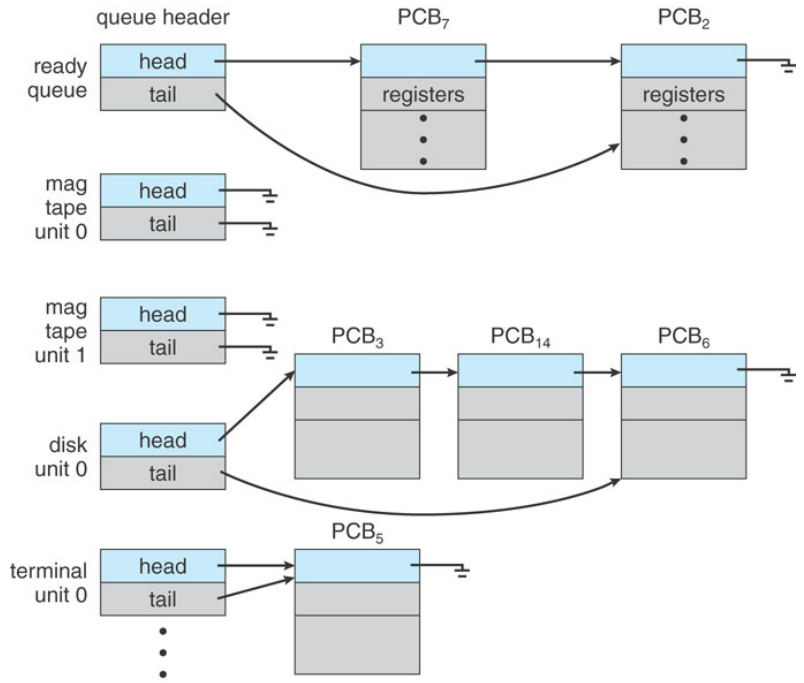
Intern in het besturingssysteem krijgt elk proces een datastructuur toegewezen die informatie bevat over de toestand van het proces. Dit procescontroleblok bevat eigenlijk drie grote stukken.

1. De contextrecord: dit is de toestand van de processor op het ogenblik dat een proces overgaat van de toestand in uitvoering naar een andere toestand. Het is deze toestand die opnieuw in de processor moet ingeladen worden om het proces weer verder te kunnen zetten. De contextrecord moet minstens plaats bevatten om de processorregisters bij te houden (de registers voor algemeen gebruik, de programmateller, toestand van de geheugenbeheerseenheid, enz.),
2. De systeemmiddelen van het proces: hoeveel geheugen werd er gealloceerd, welke bestanden en welke signalen zijn er in gebruik,
3. Beheersinformatie: de toestand waarin het zich bevindt, de identificatie van het proces, de hoeveelheid reeds gebruikte rekentijd, het ouderproces, planningsinformatie, de eigenaar van het proces.

Bemerk dat de PCB niet continu dient bijgewerkt te worden. In veel gevallen volstaat het bij de overgang naar een andere toestand de inhoud van de PCB bij te werken. In het bijzonder zal tijdens de uitvoering van het proces de contextrecord geen zinvolle informatie bevatten. Enkel bij het verlaten van de toestand in uitvoering zal de inhoud bijgewerkt worden.

Bij elke overgang zal uiteraard het veld procestoestand moeten bijgewerkt worden.

# Klaarlijst en de apparaatlijsten




best2-6

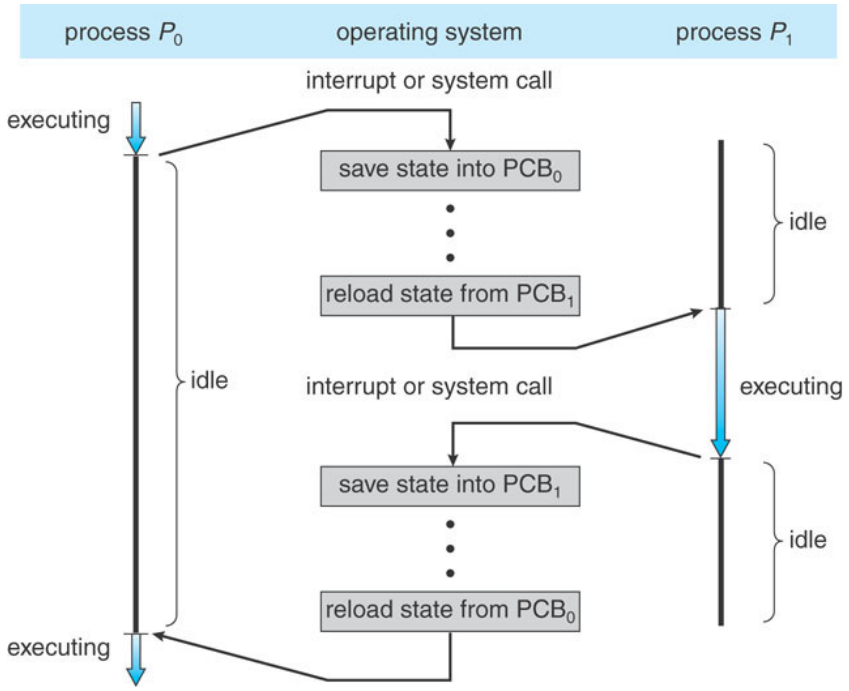
De procescontroleblokken worden gebruikt om per procestoestand een lijst van processen bij te houden. Zo bestaat de klaarlijst uit de lijst van processen in de toestand klaar. Deze lijst kan georganiseerd zijn als een gelinkte lijst, maar ook andere organisaties zijn mogelijk. In de geblokkeerde toestand kunnen verschillende deellijsten bijgehouden worden, een per type gebeurtenis waarop gewacht wordt. Op die manier zullen alle processen die staan te wachten op een bepaald randapparaat automatisch gegroepeerd en geordend worden.

Als het apparaat dan te kennen geeft dat een proces bediend kan worden, wordt de lijst overlopen op zoek naar een geschikt proces. Van dat proces wordt de toestand veranderd, en het proces wordt dan in de corresponderende lijst opgenomen. Door met gelinkte lijsten te werken volstaat het om pointers te overschrijven. Het PCB hoeft in wezen niet van plaats te veranderen.

Het PCB is een datastructuur die in de kern leeft. De kern kan op willekeurige manier velden in het PCB veranderen. Soms heeft een gebruikersproces ook nood aan de informatie die in het PCB opgeslagen ligt (b.v. om te weten te komen over hoeveel gealloceerd geheugen een proces kan beschikken). Het ter beschikking stellen van de PCB pointer aan het gebruikersproces is onaanvaardbaar omdat op die manier de afscherming van de kern zou gecompromitteerd worden. Bovendien zou het ook niet eenvoudig zijn omdat sommige onderdelen van de kern met een afzonderlijke adresruimte werken. In de praktijk zal men een identificatie van het PCB aan het gebruikersproces geven. Met die identificatie (die vaak een ‘handle’ genoemd wordt) kan

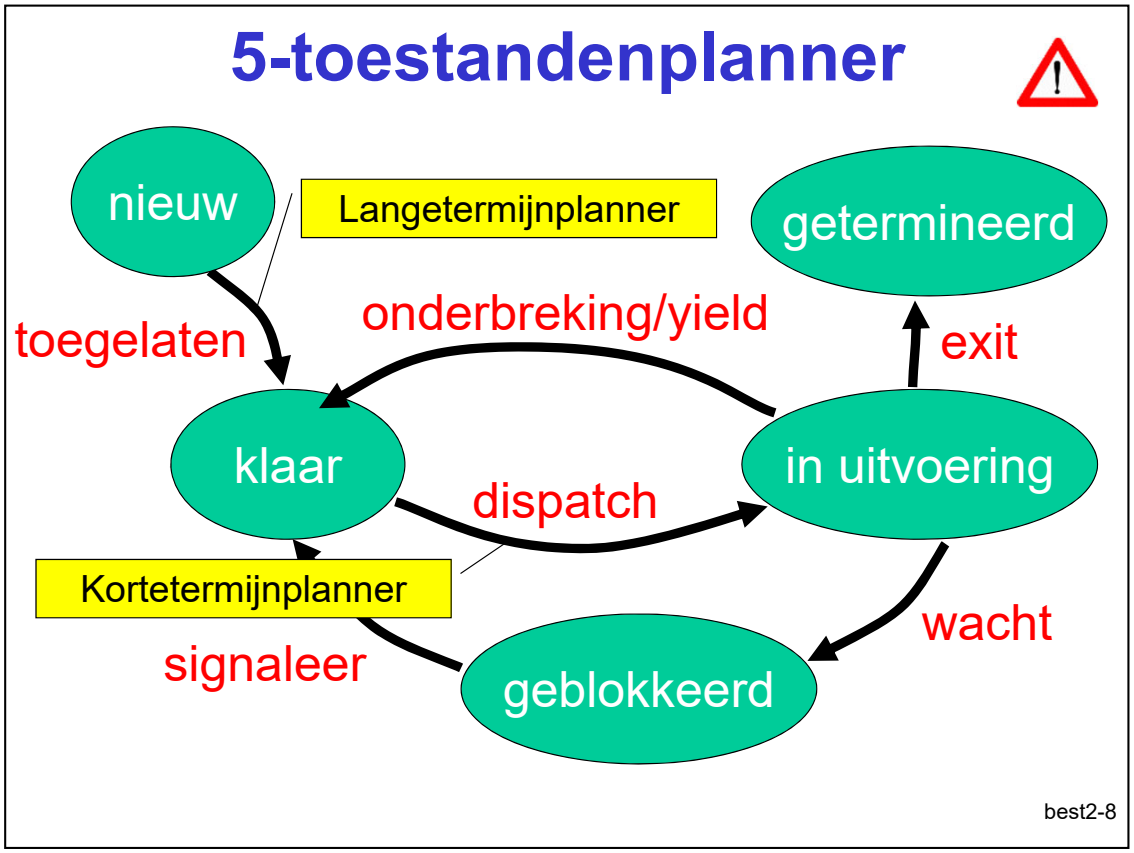
het gebruikersproces aan de kern de gewenste informatie opvragen (via een afzonderlijke systeemoproep). Het gebruik van handles is een klassieke techniek om gebruikersprocessen toegang te geven tot de kerndatastructuren.

# Contextwisseling



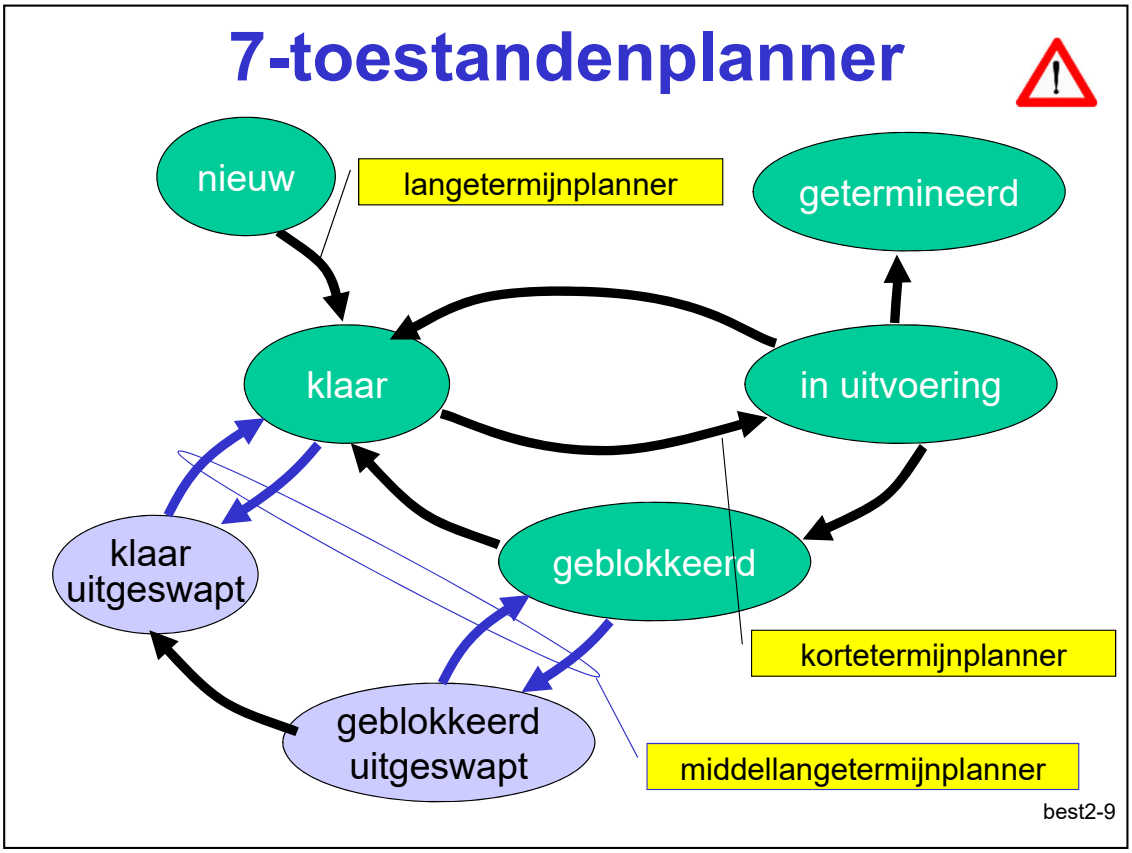

Het bewaren van het PCB van een proces en het inladen van een ander proces wordt een contextwisseling genoemd. In de praktijk komt een contextwisseling vrij vaak voor (b.v. 100 keer per seconde). Dit wil zeggen dat een proces niet langer dan 10 ms na elkaar kan uitgevoerd worden. Gelukkig kunnen er tegenwoordig meer dan 10 miljoen instructies in die 10 ms uitgevoerd worden waardoor op microscopische schaal een contextwisseling toch niet om de haverklap gebeurt. Anderzijds is een contextwisseling wel een zeer ingrijpende gebeurtenis. Het proces wordt van de processor weggehaald, en vervangen door een ander proces. De processor is in de praktijk helemaal niet voorbereid op dit nieuwe proces: de caches en de sprongvoorspeller bevatten niet de gepaste informatie. Het duurt een tijdje alvorens de processor zich aangepast heeft aan de nieuwe toestand. Jammer genoeg zal er na 10 ms opnieuw beslist worden om van proces te veranderen en herbegint het hele verhaal opnieuw.

Een contextwisseling is pure overhead, en de kost is niet verwaarloosbaar. Contextwisselingen zijn echter noodzakelijk om (i) de processor te beschermen tegen b.v. oneindige lussen, (ii) om aan multiprogrammering te kunnen doen. De kunst is om een goed evenwicht te kunnen vinden tussen het aantal contextwisselingen en de graad van reactiviteit van de processen die men nastreeft. Indien er b.v. 10 simultane gebruikers te verwachten zijn, en het tijdskwantum bedraagt 10 ms, dan zal de reactietijd in het slechtste geval 10x10ms = 100ms zijn. Voor een interactieve toepassing is dit nog aanvaardbaar. Reactietijden van 1 s zijn voor een toepassing zoals tekstverwerking niet aanvaardbaar.




De langetermijnplanner – soms ook jobplanner genoemd – bepaalt wanneer een (niet-interactief) proces kan toegelaten worden tot de klaarlijst. Dit is een manier om de graad van multiprogrammering te controleren. Indien de processor reeds voor 100% nuttig gebruikt wordt, heeft het totaal geen zin om bijkomende processen tot het systeem toe te laten. Het is veel beter om de systeemmiddelen die door deze nieuwe processen zouden ingenomen worden ter beschikking te laten aan de processen die reeds aan het uitvoeren zijn. Op die manier zullen ze sneller kunnen termineren. De langetermijnplanner zal de belasting van de processor observeren om van zodra deze dat toelaat een bijkomend proces toe te laten. Daarbij kan hij kiezen uit de lijst van nieuwe processen (b.v. het proces dat reeds langst staat te wachten, of het proces dat het minst, of juist het meest rekentijd nodig zal hebben). Heel wat interactieve systemen beschikken niet over een jobplannerer omdat ze nieuwe processen meteen toelaten tot de klaarlijst. De gebruiker zou immers niet aanvaarden dat hij minutenlang moet wachten totdat b.v. zijn editor opgestart wordt omdat de processor overbelast is. De langetermijnplanner hoeft niet frequent opgeroepen te worden (om de paar seconden tot minuten) en hoeft om die reden niet superefficiënt te zijn.

De kortetermijnplanner – vaak ook CVE-planner genoemd – heeft als taak dat proces te selecteren dat als eerstvolgende proces mag uitvoeren. Hij zal daarbij gebruik maken van een planningsalgoritme dat voorrang geeft aan bepaalde soorten processen. Dit is evenwel het onderwerp van een volgende les. De kortetermijnplanner wordt na het verstrijken van elk tijdskwantum of ter gelegenheid van het optreden van een onderbreking opgeroepen. Aangezien hij typisch meer dan 100 keer per seconde opgeroepen wordt, dient hij efficiënt te zijn.




De middellangetermijnplanner werkt samen met de langetermijnplanner – en corrigeert diens fouten. Indien blijkt dat er op een bepaald ogenblik te veel actieve processen in het systeem zijn die samen meer systeemmiddelen aanvragen dan er beschikbaar zijn, dan kan het nuttig zijn om tijdelijk één of meer processen aan het systeem te onttrekken. Dat gebeurt door de inhoud van het geheugen dat ze in gebruik hebben tijdelijk te kopiëren naar de harde schijf (het zgn. uitswappen) waardoor er geheugenruimte vrijkomt voor de overblijvende processen.

In de praktijk kan men dit uiteraard enkel maar doen voor processen die zich in de toestand klaar of geblokkeerd bevinden. De beslissing om een proces uit te swappen (en ook de beslissing welk proces dit dan wel moet zijn), wordt genomen door de middellangetermijnplanner. Indien een geblokkeerd proces uitgeswapt wordt, moet het uiteraard in staat blijven om te reageren op binnenkomende signalen en om als gevolg daarvan over te gaan naar de toestand klaar-uitgeswapt.

Zolang er uitgeswapte processen zijn heeft het niet veel zin dat de langetermijnplanner nieuwe processen toelaat. Het is beter om eerst de uitgeswapte processen verder te laten uitvoeren en pas nadat alle uitgeswapte processen terug ingeswapt zijn te overwegen om nieuwe processen toe te laten.

# Programma-argumenten

```
int main(int argc, char *argv [ ])
{
  int i;
  printf(”Aantal argumenten: %d\n”, argc);
  for (i=0; i<argc; i++)
    printf(”argument %d: %s\n”, i, argv[i]);
}
```

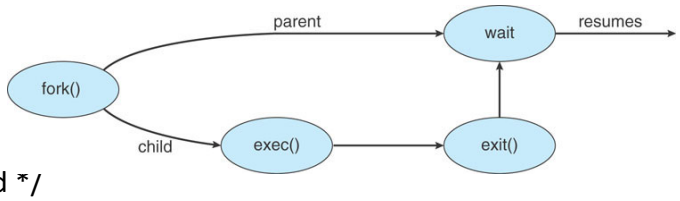
```
% cc arg.c -o arg

% arg a b c
Aantal argumenten: 4
argument 0: arg
argument 1: a
argument 2: b
argument 3: c
```

best2-10

Dit kleine programmaatje toont de argumenten waarmee het werd opgeroepen. Het eerste argument is de naam van het programma, de volgende argumenten zijn de actuele argumenten waarmee het werd opgeroepen. Het feit dat de naam van het programma als nulde argument kan opgevraagd worden zorgt ervoor dat men het gedrag van het programma eventueel kan aanpassen in functie van de naam van het bestand. Zo zou men b.v. de taalinstelling van een tekstverwerker kunnen laten afhangen van de naam waarmee het programma opgeroepen werd (word, woord, mot).

# Procescreatie in Unix (1)

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

int main(int argc, char *argv [ ])
{
    pid_t pid;
     /* fork another process */
     pid = fork();
     if (pid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed");
            exit(-1);
     }
     else if (pid == 0) { /* child process */
            execlp("./arg","arg", "a", "b", "c", NULL);
     }
     else { /* parent process */
            /* parent will wait for the child to complete */
            wait(NULL);
            printf("Child Complete");
            exit(0);
     }
}
```

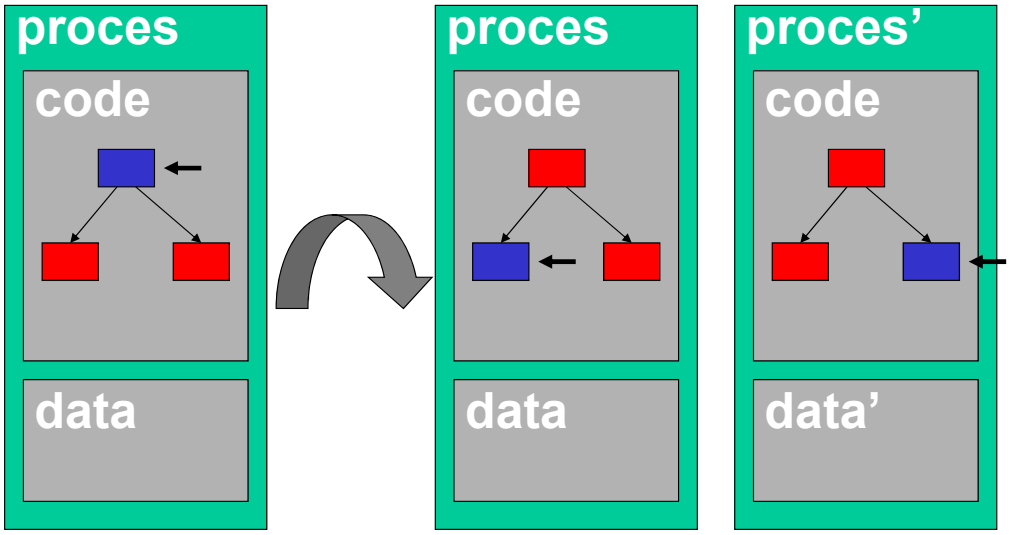



best2-11

Dit programma creëert een extra proces. Het doet dit door eerst het bestaande proces te ontdubbelen (via fork). Fork retourneert aan het ouderproces de procesidentificatie van het kindproces. Het kindproces krijgt als waarde 0 terug. Indien de teruggegeven waarde negatief is, geeft dit aan dat er bij het ontdubbelen van het proces een fout gebeurd is. Het louter ontdubbelen van een proces heeft niet veel zin. Daarom zal het kindproces in de regel meteen beslissen om het proces te overschrijven met een nieuwe applicatie. Dit gebeurt aan de hand van execlp. Het eerste argument van execlp is de naam van het bestand, de volgende argumenten zijn de argumenten zoals ze in de argv-array van het opgeroepen proces zullen verschijnen, te beginnen met de naam van het proces (die zoals hier blijkt verschillend kan gekozen worden van de naam van het bestand). De lijst wordt afgesloten met een NULL-pointer.

Via de oproep van wait kan het ouderproces blokkeren totdat het kindproces afgelopen is. Op het einde termineert het hoofdproces dan met exit(0).

# Procescreatie in Unix (2)




**Kloon overladen:** `execlp("prog",arg0, arg1,..)`

best2-12

Hier wordt nogmaals schematisch weergegeven wat fork precies doet. Het dupliceert de adresruimte van het oorspronkelijk proces. Het oorspronkelijker proces zal zijn uitvoering verder zetten als ouderproces (het proces dat van fork een waarde groter dan 0 meekrijgt), en het andere zal zijn uitvoering als kind verderzetten (het proces dat van fork de waarde 0 terugkrijgt). In het begin zijn de twee adresruimten identiek, maar uiteraard gaan de beide processen hun eigen gang waardoor ze gaandeweg meer van elkaar zullen beginnen verschillen. In eerste instantie zullen de datagebieden beginnen verschillen.

Het dupliceren van alle geheugengebieden heeft eigenlijk geen zin. Daarom zal men initieel de twee adresruimten naar hetzelfde geheugengebied laten wijzen. Enkel indien 1 van beide processen iets verandert aan een geheugengebied, zal het ontdubbeld worden. Men noemt dit copy-on-write.

Voor wat betreft de verdeling van de systeemmiddelen kunnen er zich drie scenario's voordoen: (i) ouder en kind kunnen alle systeemmiddelen delen; (ii)

kinderen kunnen toegang hebben tot een deel van de systeemmiddelen van de ouder, en (iii) ouder en kind delen geen systeemmiddelen.

# Procestoestanden in Unix

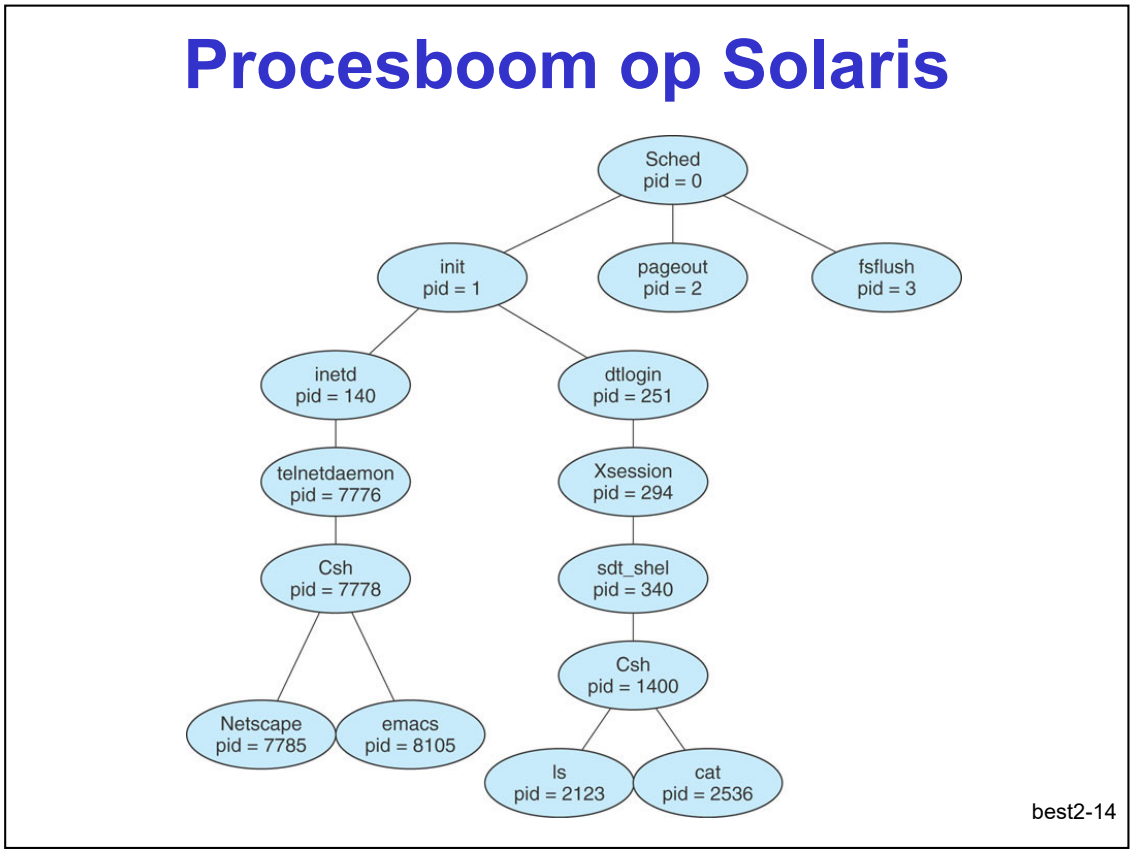
```
F S UID   PID  PPID PRI NI    ADDR   SZ    WCHAN TTY   TIME COMD
1 S 120 25718 25636  41 20 41f3480 1285   210aa0 ttypb 0:00 hpterm
1 S 120 25636 25635  41 20 4363d80   78 7ffe6000 ttypb 0:00 csh
1 R 120 25727 25636  41 20 41fac80   22          ttypb 0:00 ps
1 S 120 25722 25636  41 20 425c900   84   210aa0 ttypb 0:00 xterm
0 S 120 20480 20465   0 20 41f6900 1285   210aa0 ?     0:00 console
```

best2-13

Alle processen kunnen opgevraagd worden in Unix met het commando ps. De legende is als volgt.

| | |
|---|---|
| F | (Flag) 0: swapped, 1: in geheugen |
| S | (State) S: sleeping, R: running |
| UID | (User ID) |
| PID | (Process ID) |
| PPID | (Parent Process ID) |
| PRI | (Prioriteit) lagere waarde is hogere prioriteit |
| NI | (Nice waarde) |
| ADDR | (Adres in geheugen) |
| SZ | (Size in blocks) |
| WCHAN | (Event a process is waiting on) |
| TTY | (Controlling terminal) |
| TIME | (Cumulative execution time) |
| COMD | (Command name) |

Vergelijkbare informatie kan je opvragen in Windows via de task manager.




Bij het opstarten van het OS wordt bootstrapcode uitgevoerd: het boot proces (Sched in de figuur). Het boot proces creëert nadien het init-proces dat op zijn beurt weer andere processen creëert. Alle processen vormen een gerichte acyclische graaf in Unix. In Windows behoren alle processen tot dezelfde generatie en wordt de ouder-kind relatie niet expliciet gebruikt door het besturingssysteem (uiteraard kan men die informatie wel opvragen en visualiseren indien dat gewenst zou zijn).

# Procesterminatie

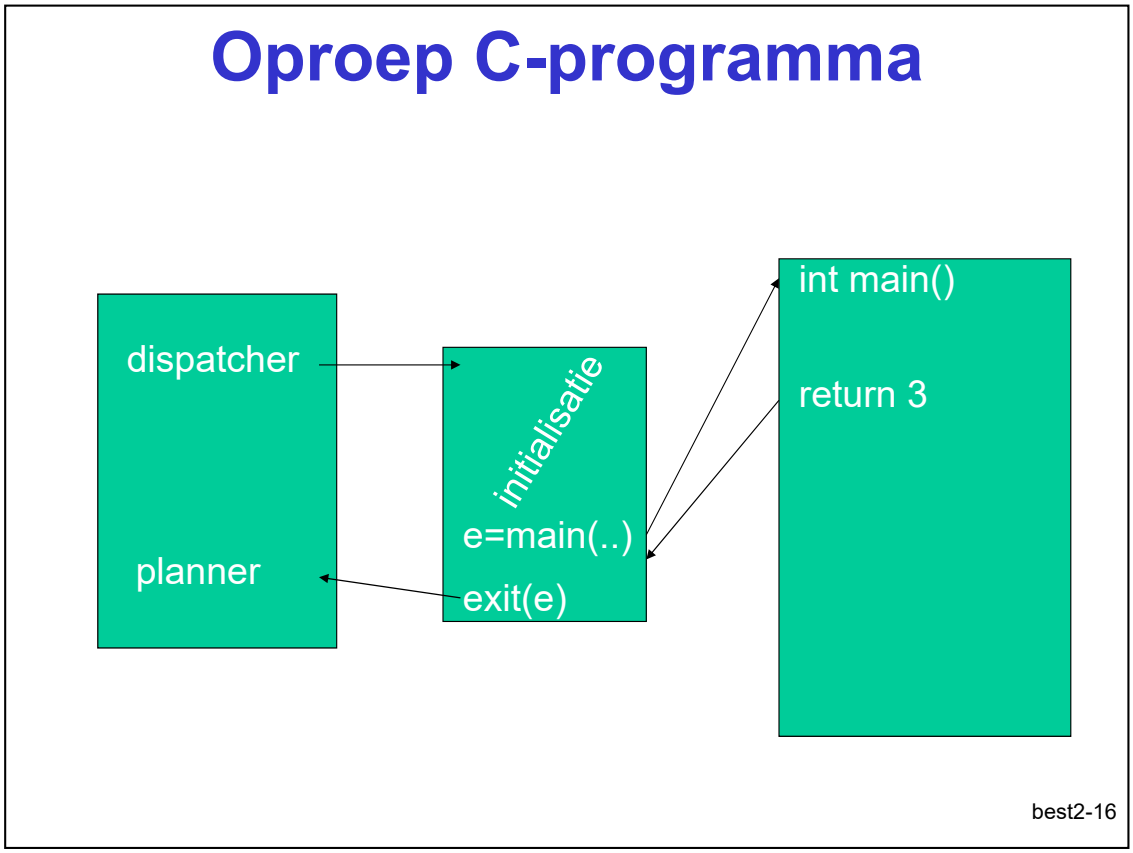
- Vrijwillig, via systeemoproep
  - werk gedaan [exit(0)]
  - fout (bvb. bestand niet gevonden) [exit(3)]
- Onvrijwillig
  - ongeldige handeling (bvb. ongeldige instructie)
  - een verzoek van ander proces (kill)
  - Cascading termination

best2-15

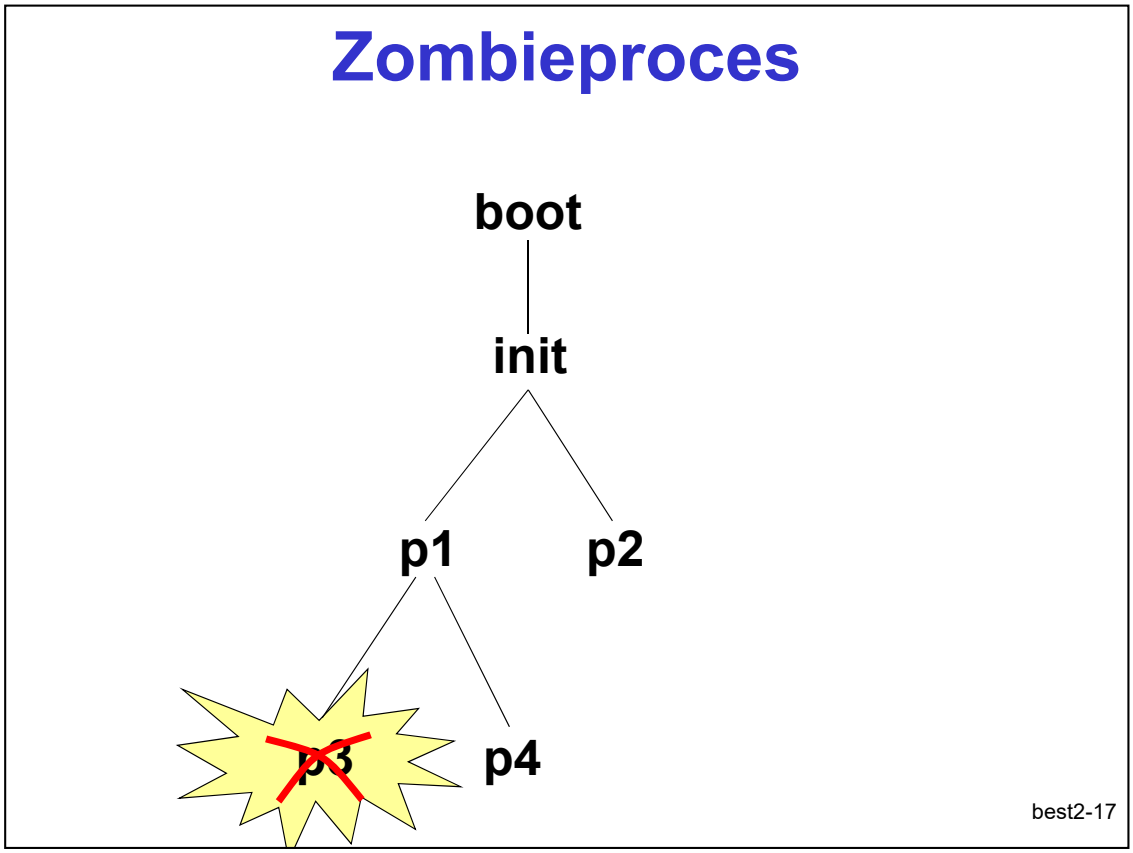
Een proces termineert door het uitvoeren van de exit systeemoproep. Dit wil zeggen dat het proces aan het besturingssysteem vraagt om getermineerd te worden. Het mechanisme van procesterminatie is m.a.w. totaal verschillend van het mechanisme van functie- of methodeterminatie (door middel van de uitvoering van een return instructie). Een proces keert m.a.w. niet terug naar het besturingssysteem, maar beslist om de controle terug over te dragen.

Ter gelegenheid van de uitvoering van de exit-systeemoproep kan er ook een getal teruggegeven worden (de exittoestand). Als deze 0 is spreken we van een normale terminatie. Als de waarde verschillend is van 0 dan spreken we over een abnormale terminatie. Deze exittoestand kan b.v. gebruikt worden in scripts om een onderscheid te maken tussen een succesvolle uitvoering van een proces en een mislukte uitvoering.

Onvrijwillige terminatie kan gebeuren als gevolg van een ongeldige handeling (b.v. nuldeling), een vraag van een ander proces om het betrokken proces te termineren, of cascading termination, dit is een situatie waarbij het ouderproces termineert, en als gevolg hiervan ook alle kindprocessen moeten termineren. Op deze manier kan een volledige procesboom getermineerd worden, ter starten met de wortel van de boom. Deze optie is niet in alle besturingssystemen aanwezig.

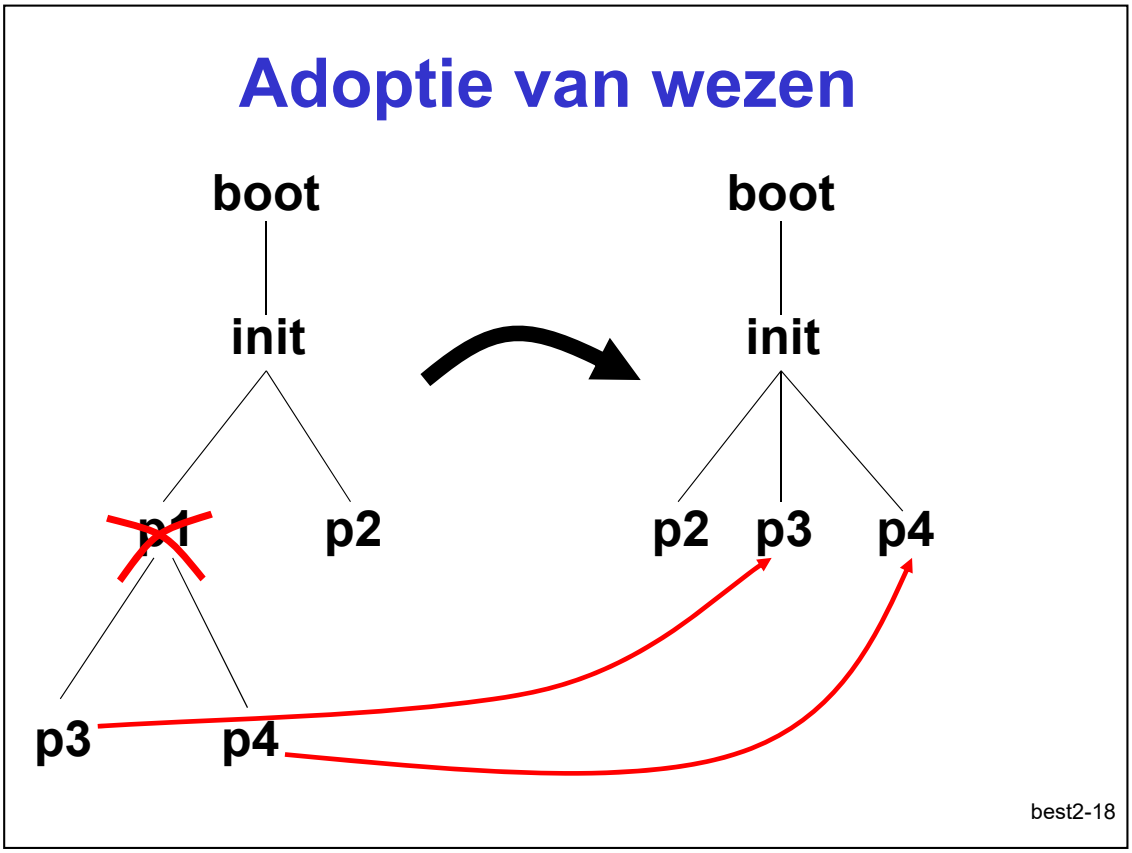



Een C-programma is eigenlijk een reguliere C-functie die eindigt met een return. Hoe kan een dergelijke functie zich uiteindelijk gaan gedragen als proces? De uitleg is heel eenvoudig. De main-functie is niet het entry-point van het C-programma. In werkelijkheid wordt de main-functie opgeroepen vanuit een ander stuk code dat o.a. de argumenten voor de main-functie klaarzet (argc, argv), en ervoor zorgt dat na het terugkeren van de main-functie toch nog een exit-oproep gebeurt, zoals vereist door het besturingssysteem. Het stuk code verantwoordelijke voor het oproepen van de main-functie wordt automatisch door de linker toegevoegd aan het programma. Dit alles blijft echter verborgen voor de C-programmeur. De return-code van de main functie wordt doorgegeven aan exit()




Een ouderproces heeft de mogelijkheid om de exittoestand van zijn kindprocessen op te vragen. Aangezien het ogenblik van termineren van het kindproces niet vastligt en van uitvoering tot uitvoering kan verschillen (b.v. omdat er nog andere processen zijn die ook competitie voeren voor de processor), zal na het termineren van het kindproces de exittoestand moeten bijgehouden worden totdat het ouderproces erom vraagt, of totdat het duidelijk wordt dat het ouderproces er nooit meer zal om vragen omdat het getermineerd werd. Van een getermineerd proces zal minstens de PCB bewaard moeten worden om exittoestand te kunnen bewaren. De overige systeemmiddelen zoals geheugen en open bestanden kunnen in principe reeds vrijgegeven worden.

Een proces waarvan de systeemmiddelen reeds vrijgegeven werden, maar waarvan de PCB nog bestaat wordt een zombieproces genoemd. Tussen de tijd dat p3 termineert en p1 de exitwaarde opvraagt zal p3 als zombieprocess in de procestabel blijven staan. Als bij het termineren van p3, p1 reeds stond te wachten, dan wordt p3 uiteraard compleet getermineerd en ontstaat er geen zombieprocess.




Een probleem dat zich kan voordoen in een proceshiërarchie is dat een ouderproces termineert terwijl de kinderen verder blijven bestaan (in geval er geen cascading terminatie is). Het laten bestaan van de weesprocessen zonder ouder kan bij hun terminatie problemen veroorzaken omdat zij tevergeefs zullen staan wachten op het ogenblik dat het ouderproces de exittoestand opvraagt. Om dat probleem op te lossen worden weesprocessen automatisch toegewezen aan het initproces.

Om het voorgaande probleem op te lossen zal het initproces ononderbroken de exittoestand van zijn kindprocessen opvragen zodat ze na hun terminatie meteen ook hun PCB kunnen vrijgeven.

# Procescreatie: Win32

```
BOOL CreateProcess(
   LPCTSTR lpApplicationName,
   LPTSTR lpCommandLine,
   LPSECURITY_ATTRIBUTES lpProcessAttributes,
   LPSECURITY_ATTRIBUTES lpThreadAttributes,
   BOOL bInheritHandles,
   DWORD dwCreationFlags,
   LPVOID lpEnvironment,
   LPCTSTR lpCurrentDirectory,
   LPSTARTUPINFO lpStartupInfo,
   LPPROCESS_INFORMATION lpProcessInformation
);
```

best2-19

In Win32 gebeurt de creatie van processen op een iets meer gestructureerde manier. De systeemoproep CreateProcess maakt een nieuw proces aan, en verwacht hierbij alle parameters die van belang zijn bij het creëren van een nieuw proces zoals de naam van de applicatie, de commandolijn waarmee het proces opgeroepen werd, beveiligingsattributen van het proces en van de draden, of het proces al dan niet de handles (systeemmiddelen) van het ouderproces overerft, de vlaggen die aangeven op welke manier het proces gecreëerd moet worden, de omgevingsveranderlijken, de directory waarin de applicatie zal uitgevoerd worden, enz.

Deze aanpak is duidelijk heel wat gestructureerder dan de Unix aanpak die het resultaat is van 30 jaar evolutie.

# Procescreatie in Win32

```
#include <windows.h>

int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow){
   STARTUPINFO si;
   PROCESS_INFORMATION pi;

   // allocate memory
   ZeroMemory(&si, sizeof(si));
   si.cb = sizeof(si);
   ZeroMemory(&pi, sizeof(pi));

   //create child process
   if (!CreateProcess(NULL, // use command line
                  "C:\\WINDOWS\\system32\\mspaint.exe", // command line
                  NULL,  // don't inherit process handle
                  NULL,  // don't inherit thread handle
                  FALSE, // disable handle inheritance
                  0,     // no creation flags
                  NULL,  // use parent's environment block
                  NULL,  // use parent's directory
                  &si,
                  &pi)) {
      MessageBox(NULL, "Create Process Failed", "BS - Process creation", MB_OK);
      return -1;
   }
    // parent will wait for the child to complete
    WaitForSingleObject(pi.hProcess, INFINITE);
    MessageBox(NULL, "Child Complete", "BS - Process creation", MB_OK);
    CloseHandle(pi.hProcess); CloseHandle(pi.hThread);
    return 0;
}
```

best2-20

Voorbeeld van procescreatie. De logica is volledig analoog met de procescreatie in Unix.

# Procescreatie in Java

```
import java.io.*;

public class OSProcess
{
   public static void main(String[] args) throws IOException {
      if (args.length != 1) {
         System.err.println("Usage: java OSProcess <command>");
               System.exit(0);
      }
      // args[0] is the command
      ProcessBuilder pb = new ProcessBuilder(args[0]);
      Process proc = pb.start();
      // obtain the input and output streams
      InputStream is = proc.getInputStream();
      InputStreamReader isr = new InputStreamReader(is);
      BufferedReader br = new BufferedReader(isr);
      String line;
      while ( (line = br.readLine()) != null)
         System.out.println(line);
         br.close();
   }
}
```

best2-21

De creatie van een (extern) proces in Java gebeurt via de ProcessBuilder klasse. De methode getInputStream van het gecreëerde proces capteert de output van het opgestarte proces.

# Commandoshell (1)

```
#include <stdio.h>
#include <sys/wait.h>
#include <string.h>

main()
{
  char commando[256];

  while ((printf("@ "), gets(commando))) {
    char *argv[16];
    int argc = 0;
    int pid;
```

best2-22

Als illustratie van procescreatie tonen we nu de implementatie van een eenvoudige commandoshell in Unix. Op deze slide wordt een commandolijn ingelezen.

# Commandoshell (2)

```
/* parsen van een commando */
argv[0] = strtok(commando," ");
while (argv[++argc] = strtok(NULL," "))
  ;

/* uitvoeren van een commando */
if (argv[0] == NULL) {
  /* doe niets */
} else if (strcmp(argv[0],"stop") == 0) {
  printf("\n");
  exit(0);
} else
```

best2-23

Aan de hand van de functie strtok wordt de commandolijn in individuele elementen opgebroken die opgeslagen worden in de array argv.

Indien de commandolijn leeg is gebeurt er niets, indien ze het woord ‘stop’ bevat wordt de commandoshell getermineerd.

# Commandoshell (3)

```
{
  int synchroon;
  if (strcmp(argv[argc-1],"&") == 0) {
    synchroon = 0;
    argc--;
    argv[argc] = NULL;
  } else
    synchroon = 1;
  if (pid = fork()) {
    if (synchroon) {
      int toestand;
      waitpid(pid, &toestand, 0);
    }
  } else
```

best2-24

In alle andere gevallen gaan we ervan uit dat er een nieuw proces moet opgestart worden om het ingegeven commando uit te voeren. Om te weten of we moeten wachten op het resultaat van dit proces, gaan we na of de commandolijn eindigt op een ampersand (&) of niet. Indien dit het geval is, hoeven we niet te wachten, indien niet, moeten we wel wachten. Het al dan niet moeten wachten wordt bijgehouden in de variabele synchroon.

Vervolgens wordt de commandoshell gedupliceerd en zal het ouderproces een waitpid uitvoeren indien er moet gewacht worden (synchroon == 1). In dit geval wordt er gewacht op het termineren van het proces met identificatie pid. Indien er niet hoeft gewacht te worden, wordt er ook geen wait-instructie uitgevoerd.

# Commandoshell (4)

```
{
  char prog[256];
  strcpy(prog,"./"); strcat(prog,argv[0]);
  execv(prog,argv);
  strcpy(prog,"/bin/"); strcat(prog,argv[0]);
  execv(prog,argv);
  strcpy(prog,"/usr/bin/"); strcat(prog,argv[0]);
  execv(prog,argv);
  strcpy(prog, "/usr/local/bin/");
  strcat(prog, argv[0]);
  execv(prog,argv);
  printf("%s: commando niet gevonden\n",argv[0]);
  exit(-1);
}}}}
```



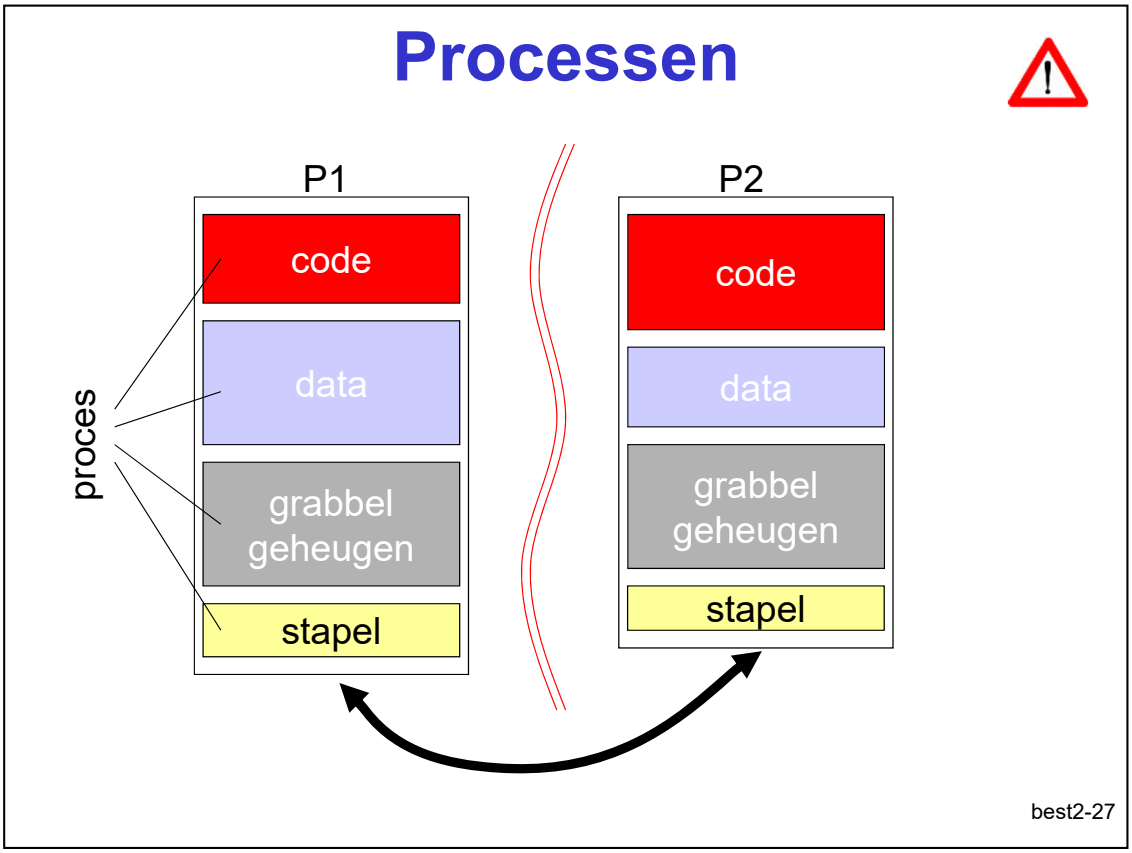
Deze slide bevat het kindgedeelte. Hier moet het proces effectief opgestart worden. Aangezien we niet zeker weten in welke directory de naam van het bestand voorkomt, zullen we een lijst met directory's aflopen. Eerst proberen we in de actieve directory (./). Als dat niet lukt (dat merken we aan het feit dat de instructie na execv uitgevoerd wordt; indien gelukt moet dit proces overschreven worden met een nieuw proces en kan deze instructie dus nooit uitgevoerd worden), dan proberen we in de directory /bin/, en vervolgens in /usr/bin/ om te eindigen bij /usr/local/bin/. Indien al deze pogingen mislukt zijn, melden we dat we het commando niet gevonden hebben. In de meeste besturingssystemen bevat de path-veranderlijke de volgorde waarin directory's moeten afgelopen worden.

Execv is een variant van execlp waarbij de argumenten als array worden doorgegeven.

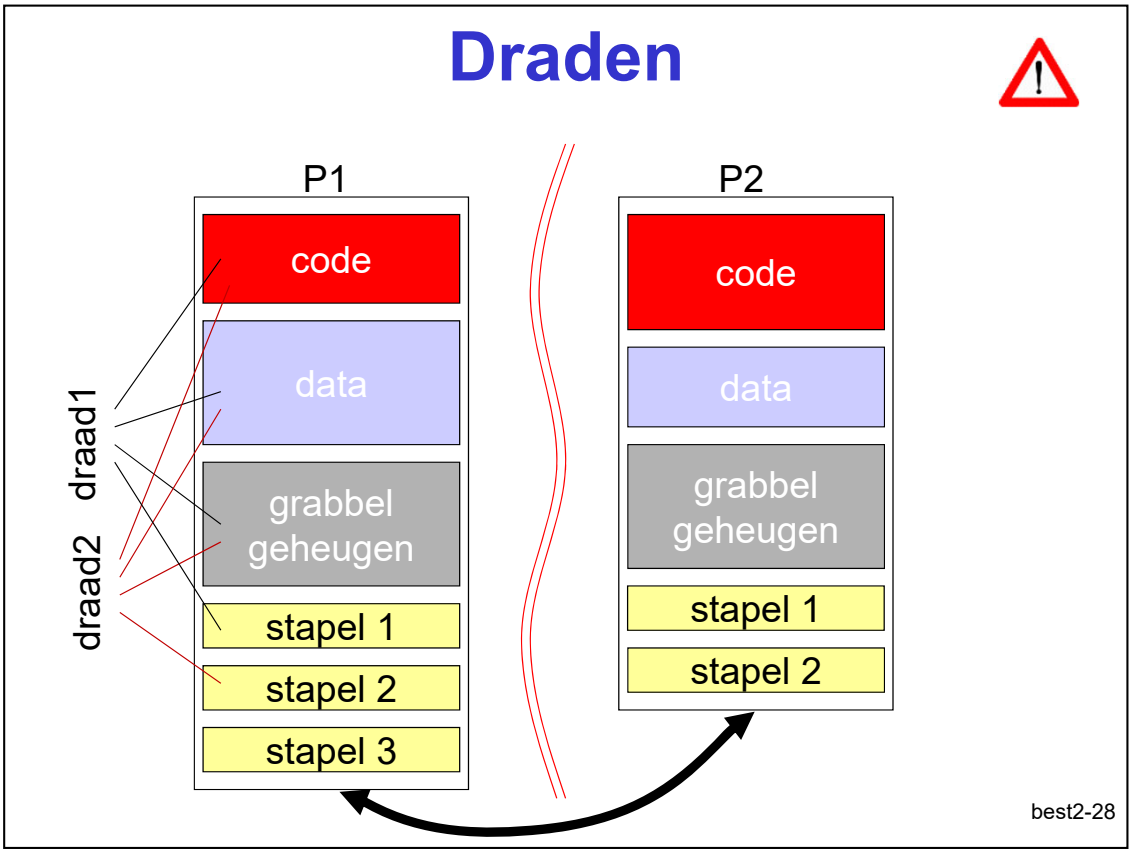
# Overzicht

- Processen
- Draden
- Voorbeelden

best2-26




Klassieke processen zijn compleet gescheiden entiteiten. Soms is het echter nuttig om een applicatie te laten bestaan uit verschillende samenwerkende processen. Door gebruik te maken interprocescommunicatieprimitieven kan men ervoor zorgen dat er gegevens kunnen uitgewisseld worden, maar echt handig is dat niet. Handiger zou zijn om een nieuw soort proces te creëren waarbij de systeemmiddelen automatisch gedeeld worden. Een dergelijk proces heet een draad (thread).

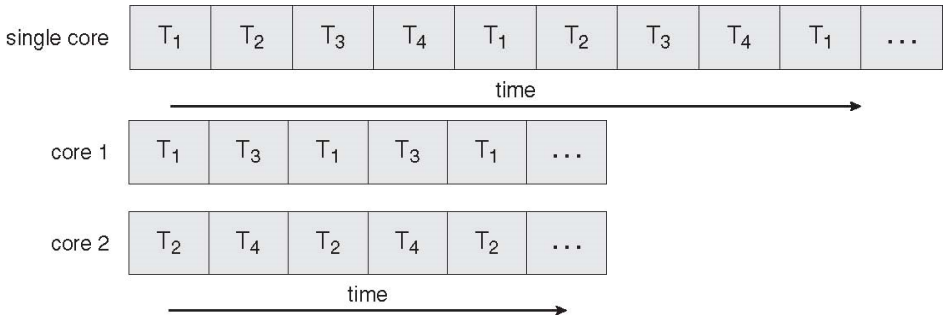



Een draad is een activiteit binnenin een proces. Een draad bestaat uit de gemeenschappelijke code, data, grabbelgeheugen en een afzonderlijke stapel. Per proces kunnen een willekeurig aantal draden aangemaakt worden. De enige kost is het alloceren van een afzonderlijke stapel voor de draad. Aangezien alle draden de code, data en grabbelgeheugen van het proces delen (evenals de open bestanden en signalen) is het voor draden zeer eenvoudig om aan een gemeenschappelijke taak te ~~werken.~~ Een proces waarin geen draden gecreëerd worden, heeft per definitie 1 enkele draad, de procesdraad. Een proces waarin slechts 1 draad actief is noemen we enkeldradig. Een proces met verschillende draden noemen we meerdradig.

~~Een proces kan beschouwd worden als een abstractie van een computer, terwijl een draad als een abstractie van een processor kan beschouwd worden. Net zoals er in een fysieke computer verschillende processors kunnen aanwezig zijn, kunnen er ook in een proces (abstracte computer) verschillende draden (abstracte processors) aanwezig zijn.~~

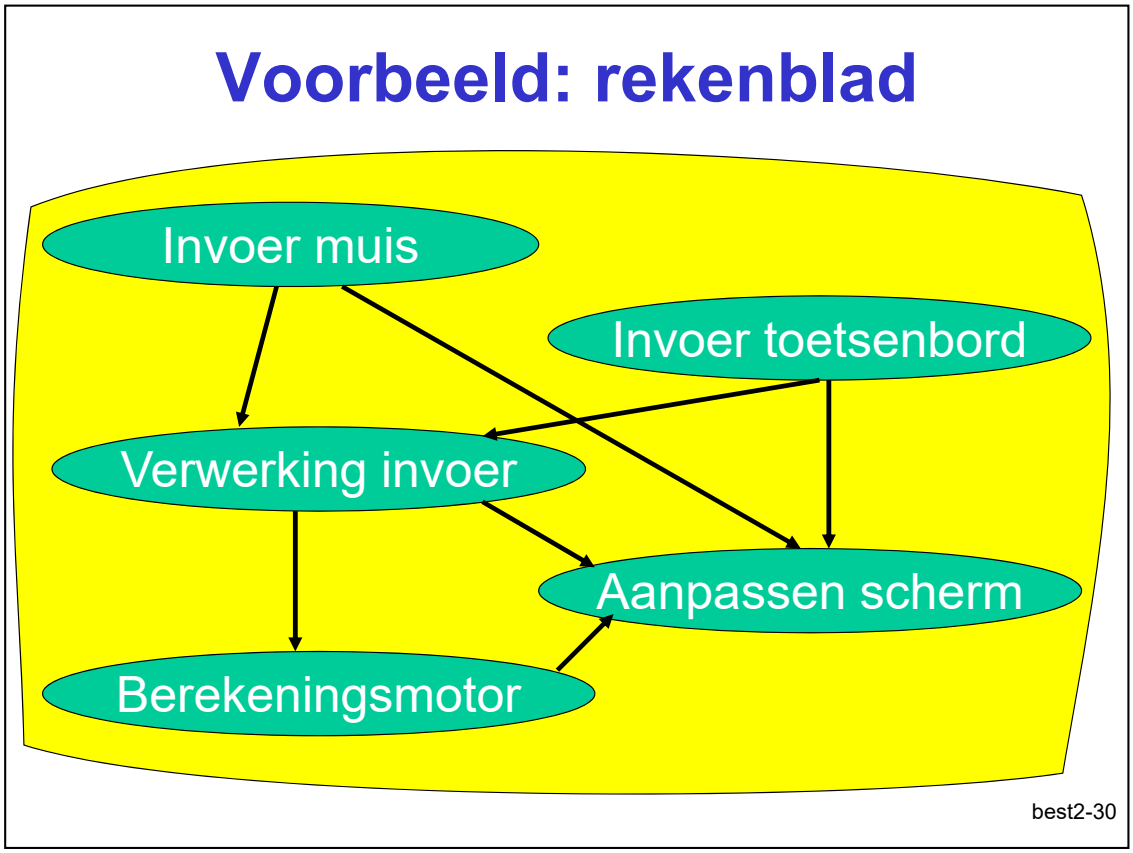
# Voordelen van draden

- Responsiviteit: programma's kunnen verder rekenen, ook indien 1 draad blokkeert
- Delen van systeemmiddelen binnenin een proces gaat automatisch
- Economischer gebruik van systeemmiddelen (door gemeenschappelijk gebruik)
- Schaalbaarheid op multicores (parallellisme)




best2-29

Deze dia somt de voornaamste voordelen van het gebruik van draden op. Om applicaties te kunnen versnellen op parallelle hardware moet men uiteraard concurrente activiteiten definiëren en daarvoor zijn draden goed geschikt. Maar zelfs indien er geen parallelle hardware in het spel is, zijn draden toch nuttig omdat ze de softwareontwikkeling kunnen vereenvoudigen zoals geïllustreerd in de volgende dia.




Het gebruik van draden maakt het programmeren van toepassingen met verschillende onafhankelijke taken een stuk eenvoudiger. Het volstaat om de diverse taken te identificeren (printen, update scherm, verwerking data) en als draad te implementeren. Een belangrijk voordeel van het werken met draden is dat wanneer er een draad blokkeert, de andere draden gewoon verder kunnen werken waardoor de responsiviteit toeneemt (in het voorbeeld zullen de draden invoer muis en invoer toetsenbord bijna altijd geblokkeerd staan; de andere draden lopen echter gewoon door). De verschillende draden zullen door middel van synchronisatieoperaties elkaar op de hoogte brengen van eventuele acties die moeten ondernomen worden.

Op een multiprocessor kunnen de verschillende draden zelfs echt parallel uitgevoerd worden. In servers kan er voor elke binnenkomende vraag b.v. een draad gecreëerd worden die deze vraag dan onafhankelijk van de rest en in zijn eigen tempo verder afwerkt.

# Gebruikersdraden vs. kerndraden

- Kerndraden worden beheerd door de kern, vergelijkbaar in gedrag met een proces
  - Win32 threads
  - POSIX Pthreads
  - Mac OS X
- Gebruikersdraden worden beheerd door een gebruikersbibliotheek, kern is niet op de hoogte
  - POSIX Pthreads
  - Win32 fibers
  - Java threads – afhankelijk van de implementatie

best2-31

Kerndraden worden beheerd door de kern, net zoals processen. kerndraden kunnen blokkeren en terug vrijgegeven worden en oefenen daarbij geen invloed uit op de andere draden van het proces. Aangezien deze draden door de kern gekend zijn, gaat hun creatie met redelijk wat overhead gepaard. Indien er vaak draden aangemaakt worden die slechts een korte tijd leven, kan hieraan echter verholpen worden door de draden niet te laten termineren, maar door ze op te nemen in een zgn thread pool en ze van daaruit nieuw werk te bezorgen (zie verder).

Gebruikersdraden daarentegen worden gecreëerd door de gebruiker zelf (zonder tussenkomst van de kern), met behulp van een speciale bibliotheek. Het voordeel hiervan is dat deze draden zeer snel kunnen aangemaakt worden (geen tussenkomst van de kern vereist), dat ze bij elk OS die de bibliotheek ondersteunt kunnen toegepast worden, en doordat ze onder de volledige controle van de gebruiker vallen kunnen aangepast worden aan de noden van de toepassing. De bibliotheek is ook verantwoordelijk voor de wisseling tussen de gebruikersdraden (hetzij door het vrijwillig afstaan van de processor, hetzij door een lokaal gedefinieerde timerroutine die de draad onderbreekt). Hoe dan ook, alles moet gebeuren onder de controle van de bibliotheek, en in usermode.

Belangrijke nadelen van gebruikersdraden zijn dat (i) als een gebruikersdraad blokkeert (b.v. op input), dan blokkeren ook alle andere gebruikersdraden die van dezelfde kerndraad gebruik maken, (ii) een gebruikersdraad kan een kerndraad monopoliseren indien deze weigert om de processor af te staan (b.v. een oneindige lus), en (iii) bijkomende gebruikersdraden zullen er niet voor zorgen dat een toepassing sneller loopt. De rekentijd wordt immers toegekend aan de kerndraad, en de gebruikersdraden die

binnen deze kerndraad lopen moeten de rekentijd dan maar onder elkaar verdelen.

# Draadmodellen

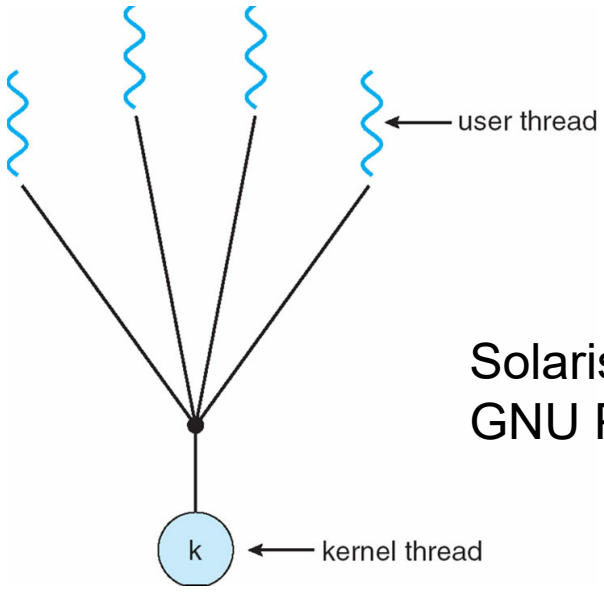
Beelden gebruikersdraden af op kerndraden

- Veel op één
- Eén of één
- Veel op veel

best2-32

Gebruikersdraden moeten steeds op de een of andere manier bebonden worden aan kerndraden om uitgevoerd te kunnen worden. Dit gebeurt aan de hand van zgn draadmodellen (multithreading models). Er zijn een drietal model die hierna besproken worden.

# Veel-op-één model

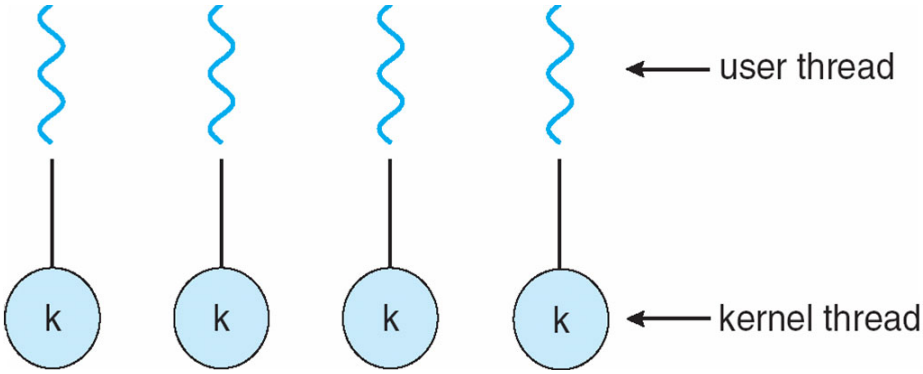



Solaris Green Threads
GNU Portable Threads

best2-33

In het veel-op-één model is er maar 1 kerndraad waarop alle gebruikersdraden lopen. Indien er 1 gebruikersdraad blokkeert, blokkeert het gehele proces. Echt parallellisme is uitgesloten. Dit model kan wel geïmplementeerd worden bovenop eender welk besturingssysteem. De kern hoeft zelfs geen kerndraden te ondersteunen. Het gewone procesmodel volstaat. Dit model wordt in de praktijk maar zelden meer gebruikt.

# Eén-op-één model



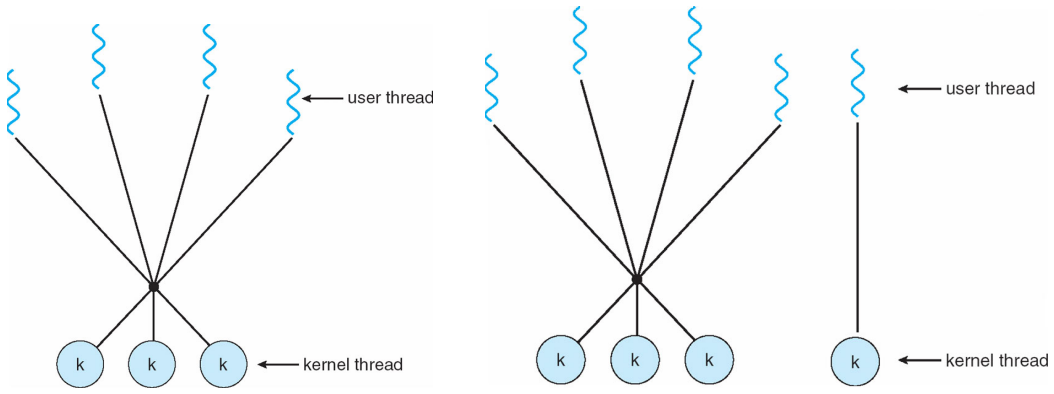

Windows
Linux
Solaris 9 en later

best2-34

Dit is het meest flexibele model, maar ook het duurste: elke gebruikersdraad heeft zijn persoonlijke kerndraad. Het ondersteunt parallellisme en de gebruikersdraden kunnen onafhankelijk van elkaar uitgevoerd wordt, ook parallel indien er verschillende processors zijn. Omdat er per gebruikersdraad een kerndraad moet opgestart worden, is dit een dure manier om gebruikersdraden te implementeren.

Dit model is op dit ogenblik het meest populaire.

# Veel-op-veel model




**Veel-op-veel**

Windows met de *ThreadFiber* package

**Tweeniveaumodel**

IRIX
HP-UX
Tru64 UNIX
Solaris < v9

best2-35

In dit model worden er een beperkt aantal kerndraden aangemaakt die dan zorgen voor de ondersteuning van een groter aantal gebruikersdraden. In het tweeniveaumodel kan men bepaalde gebruikersdraden binden aan een vaste kerndraad. In het zuivere veel-op-veel model is de binding vrij. Als er door geblokkeerde gebruikersprocessen te veel kerndraden geblokkeerd geraken, dan kan de kern beslissen om bijkomende kerndraden aan te maken om op die manier echt parallelle uitvoering op een multiprocessor te blijven garanderen.

# Pthreads

- De POSIX standaard (IEEE 1003.1c) API voor draadcreatie en synchronisatie
- Specificeert enkel de API, niet de implementatie
- Zeer courant gebruikt in Unix besturingssystemen (Solaris, Linux, Mac OS X)
- Voornaamste primitieven:
  - pthread_create()
  - pthread_exit()
  - pthread_join()

best2-36

De POSIX Pthreads standaard is de de-facto standaard voor het programmeren van meerdradige programma's. Deze standaard specificeert de interface om draden te creëren en te manipuleren. Als bibliotheek zijn pthreads op de meeste platformen beschikbaar. De voornaamste primitieven zijn het opstarten en termineren van een draad, en het wachten op een draad. De standaard is echter veel ruimer. Hun gebruik wordt geïllustreerd in de volgende slides.

# Pthreads voorbeeld

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

#define NUM_THREADS 4

void *f(void *arg) {
  int i;

  for (i=0; i<3; i++)
    printf("Uitvoering draad %d\n", *(int*)arg);
  pthread_exit(arg);
}
```

best2-37

Daar waar een proces de **uitvoering van een programma** voorstelt, stelt een draad de (asynchrone) **uitvoering van een functie** voor. In de plaats van een functie synchroon op te roepen (gebruik makende van de stapel van de oproeper, en met een oproeper die wacht totdat de opgeroepen functie terugkeert), wordt de functie nu asynchroon opgeroepen. De functie heeft een eigen stapel, en de oproeper wacht niet totdat de functie terugkeert.

Een proces met vijf draden bestaat dus uit een vijftal asynchroon uitvoerende functies, waarvan er één de main-functie zal zijn (de procesdraad).

In deze slide staat de functie f die op de volgende dia viermaal als draad zal opgestart worden. Merk op dat de functie niet langer termineert met een returninstructie, maar wel met een oproep naar pthread_exit () die de exittoestand van de draad retourneert aan die draad die erom vraagt.

# Pthreads voorbeeld

```
int main() {
 int i;
 pthread_t threads[NUM_THREADS];
 int arg[NUM_THREADS];
 void *status;

 for (i = 0; i < NUM_THREADS; i++) {
   arg[i] = i;
   if (pthread_create(&threads[i], NULL, f, (void *)&(arg[i])))
     printf("Kan draad niet aanmaken\n");
 }

 for (i = 0; i < NUM_THREADS; i++) {
   if (!pthread_join(threads[i], &status))
     printf("Draad %d gestopt met toestand %d\n",  i, *(int *)status);
 }
}
```

best2-38

In de main-functie worden de vier draden aangemaakt, ieder met hun eigen stapel (wordt automatisch gealloceerd). pthread_create()  krijgt als eerste argument een draaddescriptor terug die naderhand kan gebruikt worden om naar de draad te refereren. Het tweede argument is een structuur met attributen waarin een aantal eigenschappen van de draad kunnen gedefinieerd worden (plannereigenschappen, stapel, … hier default). Het derde argument is een pointer naar de functie die asynchroon zal uitgevoerd worden, en het laatste argument bevat een pointer naar het argument van de opgestarte functie.

Daarna zal de procesdraad wachten op de terminatie van de draden, en wordt de exittoestand van de vier draden naar het scherm geschreven.

# Linuxdraden

- Worden taken genoemd
- Geen echt onderscheid tussen processen en draden
- Worden gecreëerd door middel van clone()
- **clone()** deelt sommige systeemmiddelen met de kindtaken, aan de hand van een verzameling van vlaggen

| flag | meaning |
|---|---|
| CLONE_FS | File-system information is shared. |
| CLONE_VM | The same memory space is shared. |
| CLONE_SIGHAND | Signal handlers are shared. |
| CLONE_FILES | The set of open files is shared. |

best2-39

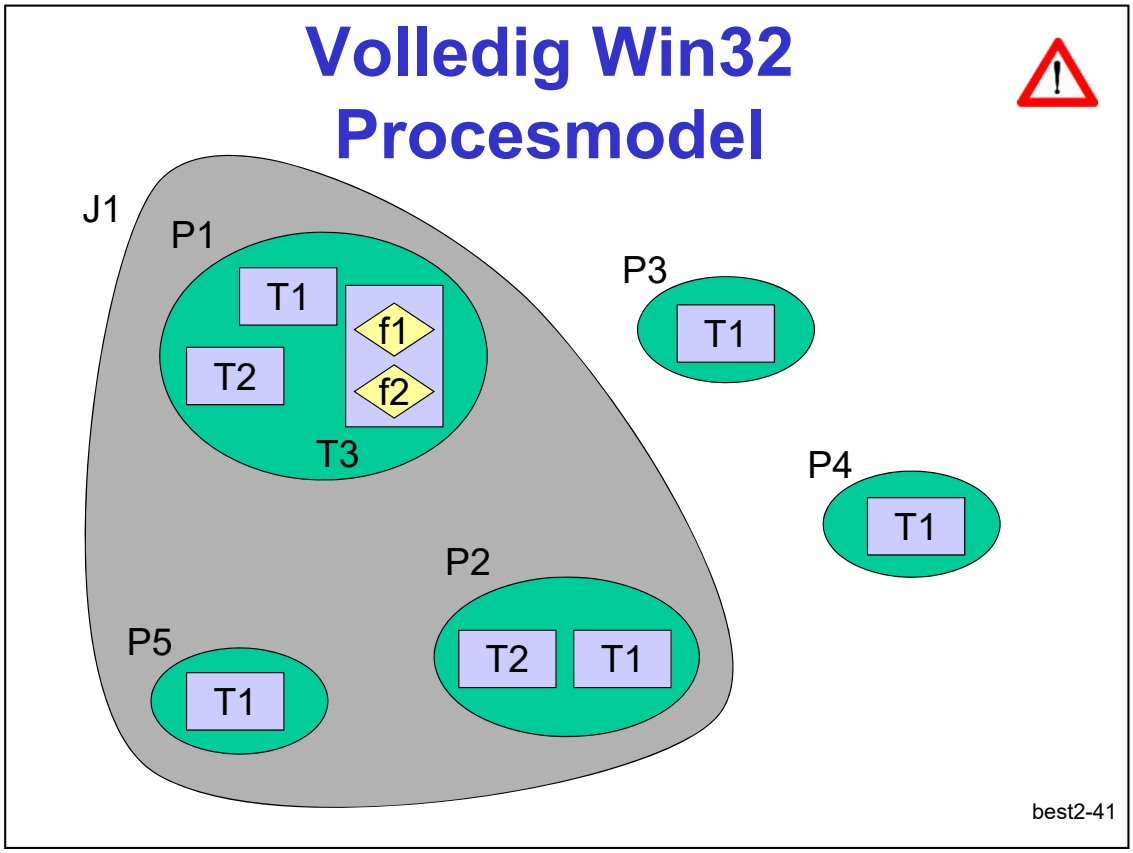
De meeste besturingssystemen implementeren de POSIXdraden bovenop de primitieven van het besturingssysteem. Zo ook voor Linux. In Linux worden draden taken genoemd, en ze worden aangemaakt met de clone systeemoproep. Naast een functie krijgt deze systeemoproep ook nog een verzameling van vlaggen mee waarvan er hier een aantal getoond worden. Deze vlaggen geven aantal welke systeemmiddelen met de kindtaak zullen gedeeld worden. Indien geen vlaggen meegegeven worden, komt clone() overeen met een fork(). Indien alle vlaggen meegegeven worden, komt het neer op een traditionele draadcreatie.

# Win32 draden

```
#include <windows.h>
#include <stdio.h>

DWORD Som = 0;

DWORD WINAPI Sommatie(LPVOID Param) {
  DWORD i;
  for (i = 0; i<*(DWORD*)Param; i++) Som += i;
  return 0;
}

int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow) {
  DWORD ThreadId;
  HANDLE ThreadHandle;
  int Param = 10;
  char txt[20];

  ThreadHandle = CreateThread(NULL,0,Sommatie,&Param,0,&ThreadId);
  if (ThreadHandle != 0) {
    WaitForSingleObject(ThreadHandle,INFINITE);
    CloseHandle(ThreadHandle);
    sprintf(txt, "Sum = %d\n", Som);
    MessageBox(NULL, txt, "BS - Threads", MB_OK);
  }
}
```

best2-40

Win32 heeft zijn eigen draadbibliotheek die vrij goed het pthreadmodel benadert.




Het basis uitvoeringsobject in Windows is het proces. In een proces kunnen verschillende draden actief zijn (dit zijn kerndraden, minstens 1). Een draad kan eventueel omgezet worden in een container voor vezels (fibers). In dat geval is de draad verantwoordelijk voor de planning van de vezels. De vezels zijn gebruikersdraden die door de applicatie zelf moeten gepland worden. Processen kunnen gegroepeerd worden in een job die bepaalde limieten kan opleggen aan de processen die tot dezelfde job behoren (b.v. maximale hoeveelheid geheugen of rekentijd).

# Javadraden

- Javadraden worden beheerd door de JVM
- Javadraden worden geïmplementeerd bovenop de primitieven die door het BS aangeboden worden (kunnen gebruikersdraden of kerndraden zijn).
- Javadraden kunnen gecreëerd worden door
  - De Thread klasse uit te breiden
  - Door de Runnable interface te implementeren

best2-42

# De Thread klasse uitbreiden

```
class Worker1 extends Thread
{
  public void run() {
    System.out.println("I Am a Worker Thread");
  }
}

public class First
{
  public static void main(String args[]) {
    Worker1 runner = new Worker1();
    runner.start();

    System.out.println("I Am The Main Thread");
  }
}
```

best2-43

De klasse Worker1 breidt de Thread klasse uit en en moet daarbij de methode run() implementeren. De method run() is de method die door de draad zal uitgevoerd worden. Deze methode wordt niet opgestart door run() op te roepen (want dan zou er geen asynchrone activiteit ontstaan), maar wel door de methode start() op te roepen. De methode start() zal een draad creëren die de methode run() van het object (asynchroon) uitvoert.

# Implementatie van Runnable

```
class Worker2 implements Runnable
{
  public void run() {
    System.out.println("I Am a Worker Thread ");
  }
}
public class Second
{
  public static void main(String args[]) {
    Runnable runner = new Worker2();
    Thread thrd = new Thread(runner);
    thrd.start();
    System.out.println("I Am The Main Thread");
  }
}
```

```
public interface Runnable
{   public abstract void run(); }
```

best2-44

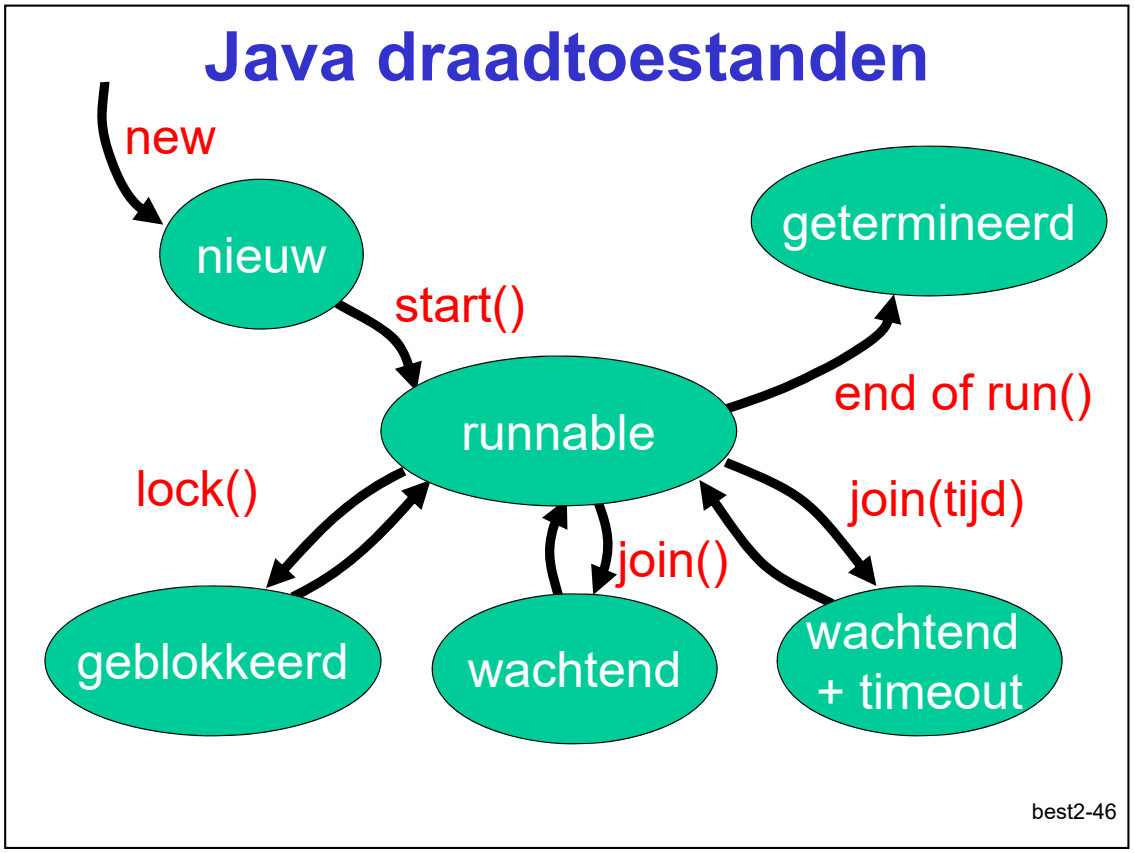
Een alternatieve manier om een draad te creëren is de implementatie van de interface Runnable zoals geïllustreerd in deze slide. Een object van de klasse die Runnable implementeert moet dan als argument bij de creatie van het Thread object meegegeven worden.

## Joining Threads

```
class JoinableWorker implements Runnable
{
  public void run() {
    System.out.println("Worker working");
  }
}

public class JoinExample
{
  public static void main(String[] args) {
    Thread task = new Thread(new JoinableWorker());
    task.start();
    try { task.join(); }
    catch (InterruptedException ie) { }
    System.out.println("Worker done");
  }
}
```

best2-45

Er kan uiteraard ook op het termineren van Javadraden gewacht worden. Men zet de join() operatie best in een try-catch blok zodat een onderbreking tijdens het wachten netjes kan afgehandeld worden.




Deze dia toont het toestandsdiagramma voor Javadraden. Draden die ‘klaar’ zijn of ‘in uitvoering’ bevinden zich steeds in de ‘runnable’ toestand – of ze nu aan het uitvoeren zijn of niet. Draden die om de een of andere reden niet uitvoeren of niet klaar zijn om uitgevoerd te worden, bevinden zich in één van de onderste toestanden, afhankelijk van het soort van signaal waarop ze staan te wachten. Een draad bevindt zich in de geblokkeerde toestand indien hij staat te wachten op een monitorsynchronisatie (zie les 4). Een draad bevindt zich in de toestand ‘wachtend´indien hij staat te wachten op een ander proces (zonder timeout, b.v. join()). Indien een draad wacht op een ander proces met een timeout (b.v. join(10) die maximaal 10 ms wacht), dan komt hij terecht in de toestand ‘wachtend+timeout’.

# proces vs draad

| Proces | Draad |
|---|---|
| Eigen adresruimte | Eigen stapel |
| Duur in aanmaak | Goedkoop in aanmaak |
| Goede bescherming | Geen bescherming |
| ~ computer | ~ processor |
| Beheert systeemmiddelen | Beheert context (registers, PC, stapel) |

best2-47

Op deze dia worden de voornaamste eigenschappen van de draden en processen nog eens op een rijtje gezet.

De relatie van een draad tot zijn proces is vergelijkbaar met de relatie tussen de bewoners van een huis en het huis zelf. Het huis zorgt voor het comfort van alle gebruikers (gemeenschappelijke verwarming, keuken, leefkamer). Sommige onderdelen van het huis zijn exclusief voorbehouden voor bepaalde bewoners (b.v. de slaapkamer). Voor het gebruik van andere lokalen moeten afspraken gemaakt worden (b.v. toilet). De activiteiten in het huis worden verricht door de bewoners, niet door het huis zelf. Een huis heeft in principe minstens 1 bewoner.

# Problemen met draden

- Semantiek van **fork()** en **exec()** systeemoproepen
- Annuleren van draden
- Afhandeling van signalen
- Thread pools
- Draadspecifieke gegevens
- Planneractivaties

best2-48

Draden zijn een mooi concept, maar niet zonder problemen. In de volgende dia's gaan een aantal van die problemen overlopen worden.

# Semantiek van fork() en exec()

- Worden alle draden gedupliceerd door fork()? Twee varianten aanbieden:
  - Indien meteen exec volgt, volstaat 1 draad
  - Anders alle draden
- Oproep van exec? Wat met verschillende actieve draden? Alle draden moeten termineren want de code verdwijnt.

best2-49

Dit een probleem dat specifiek is voor Unix. Fork() dupliceert een proces, maar wat gebeurt er met de diverse draden?

# Annuleren van draden

- Vóór de normale terminatie
- Twee aanpakken:
  - **Asynchroon annuleren:** ogenblikkelijk
  - **Uitgesteld annuleren:** draad beslist zelf wanneer te stoppen en kan alle gestarte activiteiten eerst afronden en alle resources terug vrijgeven.

best2-50

Het annuleren of het vroegtijdig stoppen van draden is een groter probleem dan de gelijkaardige operatie bij processen. Indien men een proces stopt, dan zal het besturingssysteem er finaal wel voor zorgen dat alle systeemmiddelen terug vrijgegeven worden; het proces heeft ze toch niet meer nodig.

Bij een draad is dat verschillend: de overblijvende draden moeten verder kunnen blijven werken. Indien een draad b.v. een deel van een databank vergrendelt om een aanpassing door te voeren, en tijdens die aanpassing wordt de draad gestopt, dan kan de databank zich in een inconsistente toestand bevinden, en is het ook mogelijk dat dat deel van de databank nooit meer vrijgegeven wordt of kan worden. Het stoppen van een draad is dus veel delicater dan het stoppen van een proces en moet men de nodige zorg gebeuren. Daarom is het uitgesteld annuleren een veiliger manier van annuleren dan het asynchroon annuleren.

# Uitgestelde terminatie in Java

```
Thread thrd = new Thread (new InterruptibleThread());
Thrd.start();

. . .

// now interrupt it
Thrd.interrupt();
```

best2-51

Met interrupt() kan aan een draad gevraagd worden om te termineren. Op de volgende dia wordt uitgelegd hoe een draad op een dergelijk signaal moet reageren.

# Uitgestelde terminatie in Java

```
public class InterruptibleThread implements Runnable
{
  public void run() {
    while (true) {
      /**
       * do some work for a while
       */

      if (Thread.currentThread().isInterrupted()) {
        System.out.println("I'm interrupted!");
        break;
      }
    }
    // clean up and terminate
  }
}
```

best2-52

Om uitgesteld te kunnen termineren, moet de draad regelmatig nagaan of er gevraagd werd om te termineren. Dat gebeurt aan de hand van de methode isinterrupted(). Indien dit zo is, moet de draad dan al het nodige doen om proper te termineren.

# Thread pools

- Maak een vast aantal draden aan die staan te wachten op werk, zonder te termineren op het einde.
- Voordelen:
  - Hergebruik van een draad gaat sneller dan het aanmaken van een nieuwe draad
  - Het maximaal aantal draden kan gemakkelijk beperkt worden.
- Thread pools worden standaard aangeboden in Java (Executer interface) en in Win32.

best2-53

In servers is het interessanter om per vraag een draad aan te maken ipv een proces. Echter, het telkens opnieuw aanmaken van een draad om een vraag (b.v. vraag naar een html-pagina) te beantwoorden kan relatief veel tijd in beslag nemen (in vergelijking met de rekentijd nodig om de vraag te beantwoorden). Bovendien kan bij een zware belasting het aantal draden zo hoog oplopen dat de machine overbelast geraakt.

In deze gevallen werkt men daarom vaak met een thread pool, dit is een verzameling vooraf aangemaakte draden die hergebruikt worden. Dit is sneller dan het aanmaken van nieuwe draden en het beperkt het maximaal aantal simultaan actieve draden tot een vooraf opgegeven aantal.

# Draadspecifieke gegevens

```
class Service
{
  private static ThreadLocal errorCode = new ThreadLocal();

  public static void transaction() {
     try { -- werk -- }
    catch (Exception e) {
        errorCode.set(e);
    }
  }

  public static Object getErrorCode() {
     return errorCode.get();
  }
}
```

best2-54

Indien draden gebruik maken van objecten met statische data, dan kunnen er zich problemen voordoen bij het gebruik van de statische data. Dit wordt hierboven geïllustreerd aan de hand van het bijhouden van een foutcode. Indien er maar 1 foutcode per object is, maar dit object wordt door 2 draden gebruikt, dan zou het kunnen dat de foutcode die door 1 draad veroorzaakt wordt, door de tweede draad opgevraagd wordt. Dit is uiteraard niet wenselijk. Een oplossing zou kunnen zijn om per dergelijk object een array van foutcodes te definiëren (een per draad) en die dan zelf te gaan beheren. Dat is echter omslachtig.

Het is handiger om dergelijke gegevens als draadspecifieke gegevens te declareren. De statische data moet dan gealloceerd worden als ThreadLocal object dat via set() en get() kan gebruikt worden. De get() operatie zal steeds die waarde teruggeven die door de set() operatie van dezelfde draad bewaard werd. De meeste draadbibliotheken laten het gebruik van draadspecifieke gegevens toe.

# Planneractivaties

- Gebruikersdraden die blokkeren, verhinderen het hergebruik van een kerndraad door andere gebruikersprocessen
- Dit probleem kan omzeild worden door de kern upcalls te laten genereren (na het ontvangen van een signaal en net vóór het blokkeren). De planner van de gebruikersdraden kan via deze upcalls verhinderen dat de kerndraden blokkeren door bijvoorbeeld over te gaan naar de uitvoering van een andere draad.

best2-55

# Overzicht

- Processen
- Draden
- Voorbeelden

best2-56

Sommige programmeertalen hebben draadcreatie geïncorporeerd in de programmeertaal zelf zodat een programmeur niet langer zelf de draden moeten aanmaken en beheren. Hierna volgen een paar voorbeelden.

# Fortran

```
DOALL I=1,100,1
  A[I] = 0
ENDDO
```

best2-57

Fortran is een programmeertaal die nog steeds veel gebruikt worden voor de uitvoering van numerieke algoritmen. De traditionele lus in Fortran is de do-lus. Indien alle iteraties van de do-lus onafhankelijk van elkaar zijn, dan kan men deze lus omzetten in een doall-lus. In dat geval kan de compiler beslissen op welke manier de iteraties van de lus verdeeld worden over verschillende draden. De draadcreatie is volledig transparant voor de programmeur.

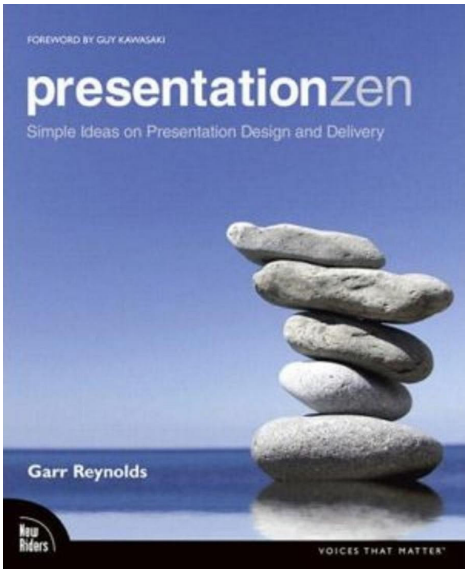
# OpenMP

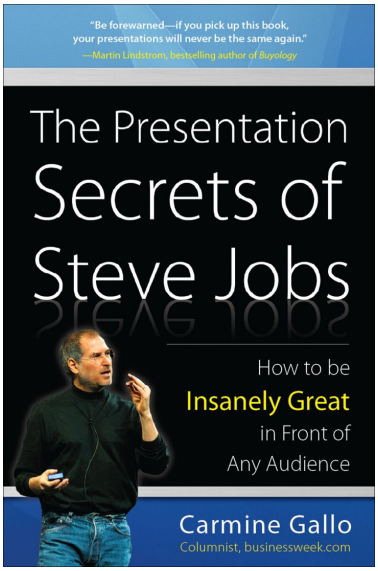
```
#pragma omp parallel for \
      private(i,j,sum) shared(m,n,a,b,c)

 for (i=0; i<m; i++) {
   sum = 0.0;
   for (j=0; j<n; j++)
     sum += b[i][j]*c[j];
   a[i] = sum;
 }
```

best2-58

OpenMP is een interface voor het parallel programmeren met gemeenschappelijk geheugen. De MP in OpenMP staat voor Multi Processing, Open betekent dat het een open standaard is, wat zoveel betekent dat iedereen er een implementatie van mag maken, zonder dat je daar één of andere instantie voor zou moeten betalen. OpenMP is gespecificeerd voor de talen C/C++ en Fortran.

OpenMP maakt gebruik van pragma's die aan de compiler vertellen op welke manier bepaalde code kan geparallelliseerd worden. In het bovenstaande voorbeeld gaat het bv over een for-lus in C. Met private wordt aangegeven welke veranderlijken moeten geprivatiseerd worden, met shared wordt aangegeven welke veranderlijken globaal zijn en eventueel moeten beschermd worden. De compiler heeft aan deze informatie genoeg om deze lus parallel uit te voeren.


FOREWORD BY GUY KAWASAKI
presentationzen
Simple Ideas on Presentation Design and Delivery
Garr Reynolds
New Riders
VOICES THAT MATTER


"Be forewarned—if you pick up this book, your presentations will never be the same again."
—Martin Lindstrom, bestselling author of Buyology
The Presentation Secrets of Steve Jobs
How to be Insanely Great in Front of Any Audience
Carmine Gallo
Columnist, businessweek.com

best2-59