2IC30: Computer systems
Translating Higher Languages

Jan Friso Groote

Technische Universiteit
Eindhoven
University of Technology

Where innovation starts

A program in machine code (PP2 processor)

Another Example

```
;
;       INPUT:   R0 = N        , precondition: 0 <= N
;       OUTPUT:  R1 = Fib(N) , R2 = Fib(N+1)
;       USES:  R3
;
```

Machine code

```
001000100 000000000
001001000 000000001
010100000 000000000
000000000 000000101
001001100 100000001
001000100 100000010
001101000 100000011
000011101 000010100
010000000 000000001
000000011 111111010
000100010 111110001
```

In PP2 assembly

```
; Rob Hoogerwoord.
;
;      INPUT:  R0 = N       , precondition: 0 <= N
;     OUTPUT:  R1 = Fib(N) , R2 = Fib(N+1)
;       USES:  R3
;
;
                 LOAD   R1    0       ;  invariant:  m = N – R0  and
                 LOAD   R2    1       ;  R1 = Fib(m) and R2 = Fib(m+1)
                 CMP    R0    0       ;  test R0, to prepare the condition codes
                 BRA         +5       ;  enter the “loop”
                 LOAD   R3   R1       ;  R2 = Fib(m+1) and R3 = Fib(m)
                 LOAD   R1   R2       ;  R1 = Fib(m+1) and R2 = Fib(m+1) and R3 = Fib(m)
                 ADD    R2   R3       ;  R1 = Fib(m+1) and R2 = Fib(m+2)
                 TRCS                 ;  abort when unsigned overflow, i.e., carry set
                 SUB    R0    1       ;  R0 := R0 - 1 , i.e.: m := m + 1 , hence:
                                      ;  R1 = Fib(m) and R2 = Fib(m+1)
                 BNE         -6       ;  if R0 = 0 then  R1 = Fib(N)
                 RTS
@END
```

☹

Assembly Language (1)

Problem

- Manually counting branch distances still is tedious and error prone
- Manually calculating memory addresses is tedious and error prone

Solution

- Use names to identify the targets of branch instructions: *labels*

Assembly Language (1)

In ARM assembly

```
; Rob Hoogerwoord;
;     INPUT:  R0 = N      , precondition: 0 <= N
;    OUTPUT:  R1 = Fib(N) , R2 = Fib(N+1)
;      USES:  R3
;
;
    Fibonacci :  MOV   R1,#0       ;  invariant:  m = N – R0  and
                 MOV   R2,#1       ;  R1 = Fib(m) and R2 = Fib(m+1)
                 CMP   R0,#0       ;  test R0, to prepare the condition codes
                 B     Fibon_while ;  enter the “loop”
     Fibon_do :  MOV   R3,R1       ;  R2 = Fib(m+1) and R3 = Fib(m)
                 MOV   R1,R2       ;  R1 = Fib(m+1) and R2 = Fib(m+1) and R3 = Fib(m)
                 ADDS  R2,R2,R3    ;  R1 = Fib(m+1) and R2 = Fib(m+2)
                 SWIOV             ;  software interrupt when (unsigned) overflow
                 SUBS  R0,#1       ;  R0 := R0 - 1, i.e.: m := m + 1 , hence:
                                   ;  R1 = Fib(m) and R2 = Fib(m+1)
  Fibon_while :  BNE  Fibon_do     ;  if R0 = 0 then  R1 = Fib(N)
Fibonacci_end :  MOV PC,LR         ;  return to invoking program
```

Memory Addressing (0)

- Suppose an integer variable is located at word 13
- Loading this integer into register R1 (say):

  MOV R0, #13

  LDR R1, [R0]
- This is still error prone! Therefore: *always* use names!
- Calling the variable x (say), it can be *bound* to its location in the data segment by means of an "EQU" pseudo-instruction:

  .equ x 13 ; now x equals 13

  .set x 13 ; equivalent to .equ x 13

  $x = 13$; also equivalent to .equ x 13
- The same instruction to load x into R1:

  MOV R0, #x

  LDR R1, [R0]
- Advantages: less error prone and better readable

Does not work for arbitrary x. Why?

Memory Addressing (0)

- Calling the variable x (say), it can be *bound* to its location in the data segment by means of an "EQU" pseudo-instruction:

  .equ x 134657463; now the address of x equals 134657463

  .set x 134657463

  x = 134657463

- The same instruction to load x into R1 (leads to an error by the assembler):

  MOV R0, #x

  LDR R1, [R0]

- The same instruction to load x into R1:

  LDR R0, =x

  LDR R1, [R0]

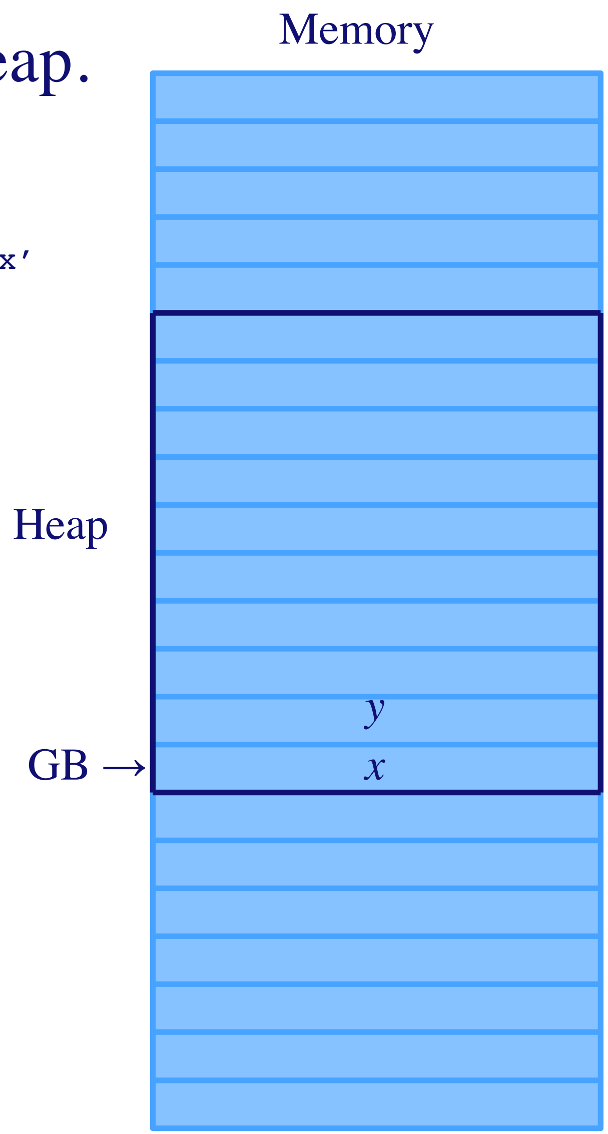
Example of a concrete assembly program.

```
.global main

.data
.equ constant 45
var_x:     .int       10
var_y:     .byte      ‘A’, 0x31, 32, 0x33, 34, $35
string:    .asciz     “This is a tekst.\n”

main:      LDR R0, =var_x

           LDR R3, [R0]
           STR  R3, [SP, #-1]!
           BL   subroutine
           STR R4, [SP], #1
           STR  R3, [R4,#constant]
           B main
.end
```

Example of the use of the heap (0)

Swap values of variables *x* and *y* on the heap.

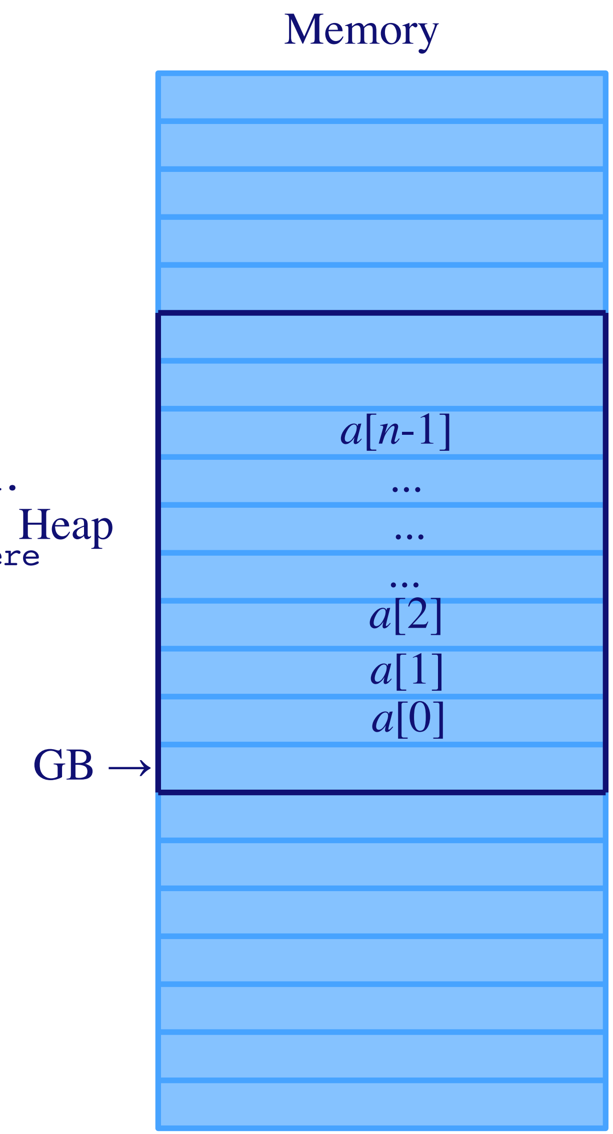
```
;   INPUT: x, y
;   OUTPUT: y is old value of x, x is old value of y: x=y', y=x'
;
global main:
main:
              LDR   GB, =global_base
              LDR   R0, [GB,#x]          ; R0:=x
              LDR   R1, [GB,#y]          ; R1:=y
              STR   R0, [GB,#y]          ; y:=R0
              STR   R1, [GB,#x]          ; x:=R1

.data
    .equ x 0                 ; x resides at GB+0
    .equ y 1                 ; y resides at GB+1
    .equ global_base  .... ; definition of the global base
```



Example of the use of the heap (1)

Reset an array $a[0,\ldots,n\text{-}1]$.

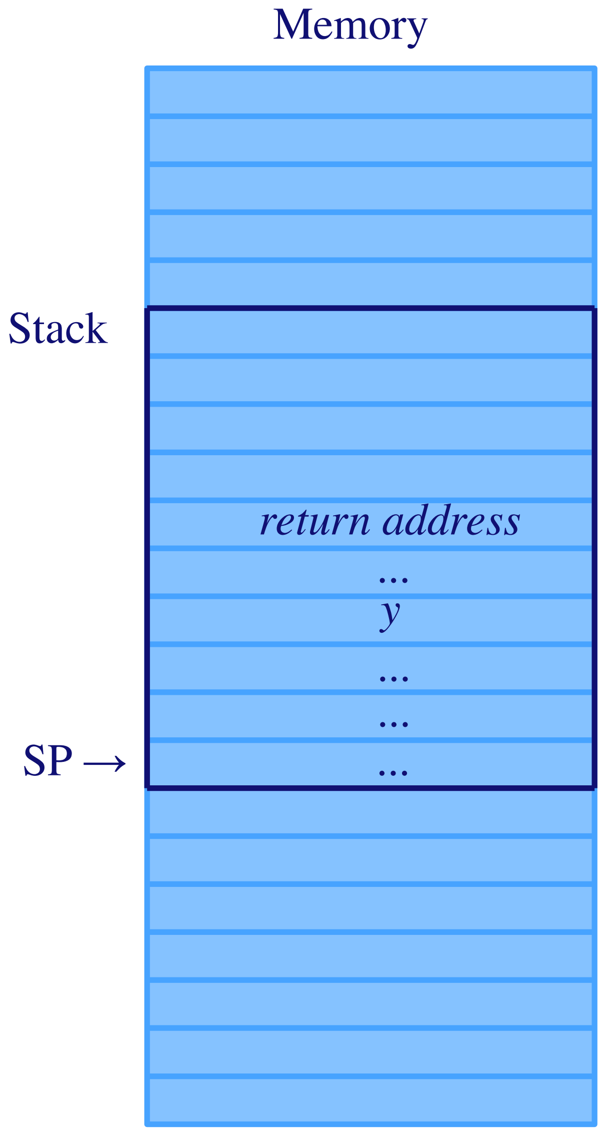
```
;  INPUT: R0, a; Precondition R0=n
;  OUTPUT: Postcondition a[i]=0 for all 0<=i<n.
;
.global main

main:
reset_a :       MOV   R1, #0          ; R1:=0 R1 represents i.
                LDR   R3, =global_base
                ADD   R3, #a          ; R3:=GB+a, address where
                                      ; array a starts.
while_reset_a: MOV   R2,R1
                SUBS  R2,R0           ; R2:=(R1-R0) R2=i-n
                BEQ   endwhile_reset  ; if i=n, stop
                MOV   R4,#0           ; R4:=0
                STORE R4 [R3,R1]      ; a[i]:=0
                ADD   R1,#1           ; i:=i+1
                B     while_reset_a   ; loop back

endwhile_reset: ...

.data
   .equ a ...                ; base address of a
   .equ global_base ...      ; definition of global base
```



Memory Addressing (1)

Local variables: used only temporarily

- Local variables can be kept in registers, or … on the *stack*!
- The stack grows *downwards* in memory; to *allocate* -- i.e. reserve -- 5 words on the stack, use: SUB SP 5.
- The addresses of these 5 words are SP, SP+1,…, SP+4.
- Better is to name them; if we use the word at position 3 as a local variable y (say) , it can be *bound* to its location in the Stack data segment by means of an ".equ" pseudo-instruction: .equ y 3; now the address of y equals SP + y.



Stack addressing: allocating local variables

- Example: A subroutine needs 6 words to store local variables.

  procedure *f*()

  var *a*, *b*, *c*, *d*, *e*, *x*: integer;

  ……

  ……

  end;

- Allocation: By *decreasing* SP by the amount –6– needed.
- Deallocation: By *increasing* SP by the amount –6– used.
- Allocation takes place at the *beginning* of the subroutine, *before* the *actual body* of the subroutine.
- Deallocation takes place at the *end* of the subroutine, *after* the *actual body* of the subroutine, but *before* the *return*.
- Every subroutine must *leave* the stack effectively *unchanged!!!*

Stack addressing: allocating local variables

```
entry:    SUB  SP  6  ;  allocate 6 words for locals
    .equ a 0          ;  “declare” local variable a
    .equ b 1          ;  “declare” local variable b
          ...
    .equ x 5          ;  “declare” local variable x
          ...
          “actual body of the subroutine”
          ...
 exit:    ADD  SP  6  ;  release the 6 words...
          RTS         ;  ... and return
```

Generation of programming languages

First generation:	Machine code
Second generation:	Assembly code
Third generation:	Higher level languages, such as ALGOL 60, Fortran, Cobol, ALGOL 68, PASCAL, C++, Java.

Prof.dr. Frans Kruseman Aretz:

"With the third level languages we thought our troubles were over. We could not be more wrong..."

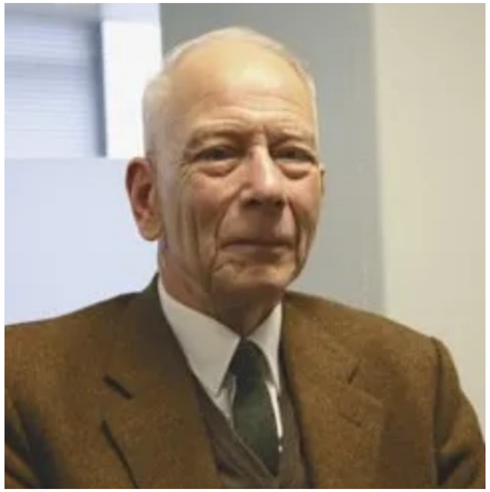


Foto: TU/e

Generation of programming languages

First generation:	Machine code
Second generation:	Assembly code
Third generation:	Higher level languages, such as ALGOL 60, Fortran, Cobol, ALGOL 68, PASCAL, C++, Java.
Fourth generation:	Not so clear. Domain specific languages??? SQL, HTML?

FORTH

APL

Large(r) Projects: Program Design

- *Separate* program design and construction of the Assembly Language program: *firstly*, design and write the program in Java, C++, … ; *secondly*, translate that program into Assembly Language.
- Why???

Large(r) Projects: Program Design

- *Separate* program design and construction of the Assembly Language program: *firstly*, design and write the program in Java, Pascal, …, or in a DSL (Domain Specific Language) or just plain mathematics; *secondly*, translate that program into Assembly Language.
- Why???

- Humans are very bad in handling complexity. Always work as abstract as as possible. Use mathematics, or any domain specific notation....

Example with recursion:

$n!$ with a recursive procedure, $n > 0$:

```
procedure fac(var n:integer);
var h:integer;
begin
   if (n≠1) then begin
      h:=n;
      n:=n-1;
      fac(n);
      n:=h*n;
   end
end;
```

In assembler:

CMP: subtract without storing the results.
MULS: multiply arguments.

```
procedure fac(var n:integer);
var h:integer;
begin
  if (n≠1) then begin
    h:=n;
    n:=n-1;
    fac(n);
    n:=h*n;
  end
end;
```

```
fac_main: CMP R0 1
          BEQ fac_exit
          STOR R0 [--SP]
          SUB R0 1
          BRS fac_main
fac_ret:  LOAD R1 [SP++]
          MULS R0 R1
fac_exit: RTS
```

Note: parameter *n* is passed in register R0.
Local variable *h* is stored on the stack.

Questions?

Technische Universiteit
Eindhoven
University of Technology

Example: Translating a Java function into Assembly (1)

$$
\begin{array}{l}
\text{int } \mathit{PowerSum}(\text{int } X, \text{int } N) \\
//\ \text{requires: } 0 \leq N \\
//\ \text{result: } \mathit{PowerSum}(X, N) = (\Sigma i : 0 \leq i \leq N : X^i) \\
\{\quad \text{int } x, y, z, n ; \\
\quad z = 0 ; y = 1 ; x = X ; n = N ; \\
\quad \text{while } (n \,!\!= 0) \\
\quad \{\quad \text{while } (n\%2 == 0) \\
\qquad\quad \{ y = y * (1+x) ; x = x * x ; n = n/2 ; \} \\
\qquad z = z + y ; y = y * x ; n = n - 1 ; \\
\quad \} \\
\quad \text{return } z ; \\
\}
\end{array}
$$

Example: Translating a Java function into Assembly (2)

(0) Decide how the *parameters* will be transferred: parameters X and N are *values* to be transferred *to* the subroutine, and the function's *result* is to be transferred back *from* the subroutine to the calling program:

- These three (integer) values are passed via the *Stack*:
- Just *before* the call of the subroutine, the calling program:

  --reserves 1 word on the stack for the function's *result*;

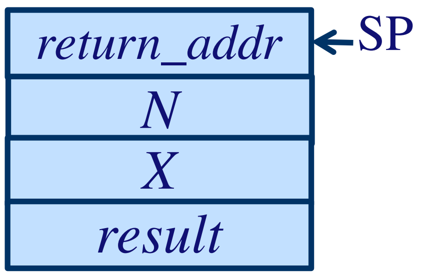
  --then, it pushes X onto the stack;

  --next, it pushes N onto the stack;

  --finally, it calls the subroutine.
- Directly *after* the call of the subroutine, the calling program:

  --retrieves 1 the function's *result* from the reserved word for this purpose;

  --then, it *cleans up* the stack;



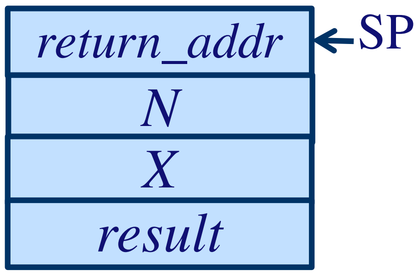
int $PowerSum$(int X, int N)

// requires: $0 \leq N$

// result: $PowerSum(X,N) = (\Sigma i : 0 \leq i \leq N : X^i)$

Example: Translating a Java function into Assembly (3)

```
;  Calling sequence to invoke subroutine PowerSum:

   call_begin :  SUB    SP   1          ;  allocate one word on the stack,
                                        ;  for the result.
                 LOAD   R0   X          ;  here we just write "X": may
                                        ;  be more complicated.
                 STOR   R0   [--SP]     ;  push X onto the stack.
                 LOAD   R0   N          ;  here we just write "N".
                 STOR   R0   [--SP]     ;  push N onto the stack.
                 BRS    PowerSum        ;  call subroutine PowerSum.
     call_end :  LOAD   R0   [SP+2]     ;  retrieve PowerSum's result...
                 STOR   R0   z          ;  ... and store it somewhere,
                                        ;  here just called "z"
                 ADD    SP   3          ;  clean up Stack: deallocate the 3
                                        ;  words used
                 ...                    ;  and continue...
```



Example: Translating a Java function into Assembly (4)

(1) When execution of the subroutine starts, the top of the stack has this d…

```
RAM[SP  ] =  "return address of the calling pro…
RAM[SP+1] =  N
RAM[SP+2] =  X
RAM[SP+3] =  "word for the (anonymous) function…
```

x	← SP
y	
n	
return_addr	
N	
X	
result, z	

(2) Now, decide where *local variables* *x, y, z, n* will be allocated. Because the final value of *z* will be returned as the result, *z* may be identified with the result word at SP+3 . Variables *x, y and n* are put on the stack:

```
x =  RAM[SP+0]
y =  RAM[SP+1]
n =  RAM[SP+2]
z =  RAM[SP+3+3]
```

Compensate for two local variables on the stack.

int $PowerSum$(int X , int N)

// requires: $0 \leq N$

// result: $PowerSum(X,N) = (\Sigma i : 0 \leq i \leq N$

{ int x , y , z , n ;

$z = 0$; $y = 1$; $x = X$; $n = N$;

while $(n != 0)$

Example: Translating a Java function into Assembly (1)

$$
\begin{array}{l}
\text{int } PowerSum(\text{int } X, \text{ int } N) \\
//\ \text{ requires: } 0 \le N \\
//\ \text{ result: } PowerSum(X,N) = (\Sigma i : 0 \le i \le N : X^i) \\
\{\ \ \text{int } x, y, z, n\ ; \\
\quad z = 0\ ;\ y = 1\ ;\ x = X\ ;\ n = N\ ; \\
\quad \text{while } (n != 0) \\
\quad \{\ \ \text{while } (n \% 2 == 0) \\
\qquad \{\ y = y * (1 + x)\ ;\ x = x * x\ ;\ n = n / 2\ ;\ \} \\
\qquad z = z + y\ ;\ y = y * x\ ;\ n = n - 1\ ; \\
\quad \} \\
\quad \text{return } z\ ; \\
\}
\end{array}
$$

Example: Translating a Java function into Assembly (5)

$$z = 0\,;\ y = 1\,;\ x = X\,;\ n = N\,;$$
$$\text{while } (n \mathrel{!}= 0)$$
$$\{\quad \text{while } (n \% 2 == 0)$$
$$\{\ y = y * (1 + x)\,;\ x = x * x\,;\ n = n/2\,;\ \}$$
$$z = z + y\,;\ y = y * x\,;\ n = n - 1\,;$$

```
.equ local_vars    3
.equ PowerSum_N    1+1
.equ PowerSum_X    2+1
.equ PowerSum_z    3+1
.equ Powersum_x    0
.equ PowerSum_y    1
.equ PowerSum_n    2
PowerSum_begin:    SUB   SP   local_vars          ; reserve room on the stack
                   LOAD  R0   0                   ; initialize variables
                   STOR  R0   [SP+PowerSum_z]     ; z ← 0
                   LOAD  R0   1                   ; R0 ← 1
                   STOR  R0   [SP+PowerSum_y]     ; y ← 1
                   LOAD  R0   [SP+PowerSum_X]     ; R0 ← X
                   STOR  R0   [SP+PowerSum_x]     ; x ← X
                   LOAD  R0   [SP+PowerSum_N]     ; R0 ← N
                   STOR  R0   [SP+PowerSum_n]     ; n ← N
PowerSum_while0:   BEQ   PowerSum_od0             ; exit outer repetition
                                                  ; flags reflect value of n
;  continued on next page...
```

```
x =  RAM[SP+0]
y =  RAM[SP+1]
n =  RAM[SP+2]
z =  RAM[SP+6]
```

Example: Translating a Java function into Assembly (6)

```
PowerSum_while1: LOAD R0 [SP+PowerSum_n]  ; R0 ← n
                 AND  R0  1               ; R0 ← n%2
                 BNE   PowerSum_od1       ; IF not n%2=0, THEN exit
                                          ; inner repetition
PowerSum_do1:    LOAD R0 [SP+PowerSum_x]  ; R0 ←  x
                 ADD  R0  1               ; R0 ←  x + 1
                 LOAD R1 [SP+PowerSum_y]  ; R1 ←  y
                 MULS R1  R0              ; R1 ←  y × (x+1)
                 STOR R1 [SP+PowerSum_y]  ; y  ←  R1
                 LOAD R0 [SP+PowerSum_x]  ; R0 ←  x
                 LO
                 MU
                 ST
                 LO
                 SH
                 ST
                 BR
                                          ; inner repetitionPowerSum_od1
; continued
```

```
x =  RAM[SP+0]
y =  RAM[SP+1]
n =  RAM[SP+2]
z =  RAM[SP+6]
```

$$z = 0\ ;\ y = 1\ ;\ x = X\ ;\ n = N\ ;$$

while $(n != 0)$

$\{$ while $(n \% 2 == 0)$

$\quad \{\ y = y * (1 + x)\ ;\ x = x * x\ ;\ n = n/2\ ;\ \}$

$\quad z = z + y\ ;\ y = y * x\ ;\ n = n - 1\ ;$

Example: Translating a Java function into Assembly (6)

```
PowerSum_while1: LOAD
                 AND
                 BNE

PowerSum_do1:    LOAD
                 ADD
                 LOAD
                 MULS R1  R0                    ; R1 ←  y × (x+1)
                 STOR R1 [SP+PowerSum_y]        ; y  ←  R1
                 LOAD R0 [SP+PowerSum_x]        ; R0 ←  x
                 MULS R0 R0                     ; R0 ←  x × x
                 STOR R0 [SP+PowerSum_x]        ; x  ←  R0
                 LOAD R0 [SP+PowerSum_n]        ; R0 ←  n
                 SHIFTRIGHT R0                  ; n ←  n/2
                 STOR R0 [SP+PowerSum_n]        ; n  ←  R0
                 BRA  PowerSum_while1           ; jump back to beginning of
                                                ; inner repetitionPowerSum_od1
```

$$
\begin{array}{l}
z = 0\,;\; y = 1\,;\; x = X\,;\; n = N\,; \\
\text{while}\ (n != 0) \\
\{\quad \text{while}\ (n\%2 == 0) \\
\qquad \{\ y = y*(1+x)\,;\; x = x*x\,;\; n = n/2\,;\ \} \\
\qquad z = z+y\,;\; y = y*x\,;\; n = n-1\,;
\end{array}
$$

```
x =  RAM[SP+0]
y =  RAM[SP+1]
n =  RAM[SP+2]
z =  RAM[SP+6]
```

```
; continued
```

Examp

$$z = 0 ;\ y = 1 ;\ x = X ;\ n = N ;$$
$$\text{while } (n != 0)$$
$$\{ \quad \text{while } (n\%2 == 0)$$
$$\{\ y = y*(1+x) ;\ x = x*x ;\ n = n/2 ;\ \}$$
$$z = z+y ;\ y = y*x ;\ n = n-1 ;$$

```
PowerSum_while1:

PowerSum_do1:    ...
                 ...
PowerSum_od1:    LOAD R0 [SP+PowerSum_z]   ; R0 ←  z
                 ADD  R0 [SP+PowerSum_y]   ; R0 ←  z + y
                 STOR R0 [SP+PowerSum_z]   ; z ←  R0
                 LOAD R0 [SP+PowerSum_y]   ; R0 ←  y
                 LOAD R1 [SP+PowerSum_x]   ; R1 ←  x
                 MULS R0  R1               ; R0 ←  y × x
                 STOR R0 [SP+PowerSum_y]   ; y ←  R0
                 LOAD R0 [SP+PowerSum_n]   ; R0 ←  n
                 SUB  R0 1                 ; R0 ←  n – 1
                 STOR R0 [SP+PowerSum_n]   ; n ←  R0
                 BRA   PowerSum_while0     ; jump back to beginning of
                                           ; outer repetition
```

```
x =  RAM[SP+0]
y =  RAM[SP+1]
n =  RAM[SP+2]
z =  RAM[SP+6]
```

Example: Translating a Java function into Assembly (8)

$$\{\ y = y * (1 + x)\ ;\ x = x * x\ ;\ n = n/2\ ;\ \}$$
$$z = z + y\ ;\ y = y * x\ ;\ n = n - 1\ ;$$
$$\}$$
$$\text{return } z\ ;$$

```
PowerSum_while1: LOAD
                 AND
                 BNE

PowerSum_do1:    ...
                 BRA    PowerSum_while1     ; jump back to the inner loop
                                            ; inner repetition
PowerSum_od1 :   ...
                 ...
                 BRA    PowerSum_while0     ; jump back to the
                                            ; outer repetition
 PowerSum_od0 :  ADD SP local_vars          ; remove local vars
                 RTS                        ; result value is already
                                            ; there: just RETURN
```

```
x =  RAM[SP+0]
y =  RAM[SP+1]
n =  RAM[SP+2]
z =  RAM[SP+6]
```

Questions?

Software interrupts or software exceptions.

- SWI #number.   ; ARM processor.
- TRAP   ; PP2 processor.

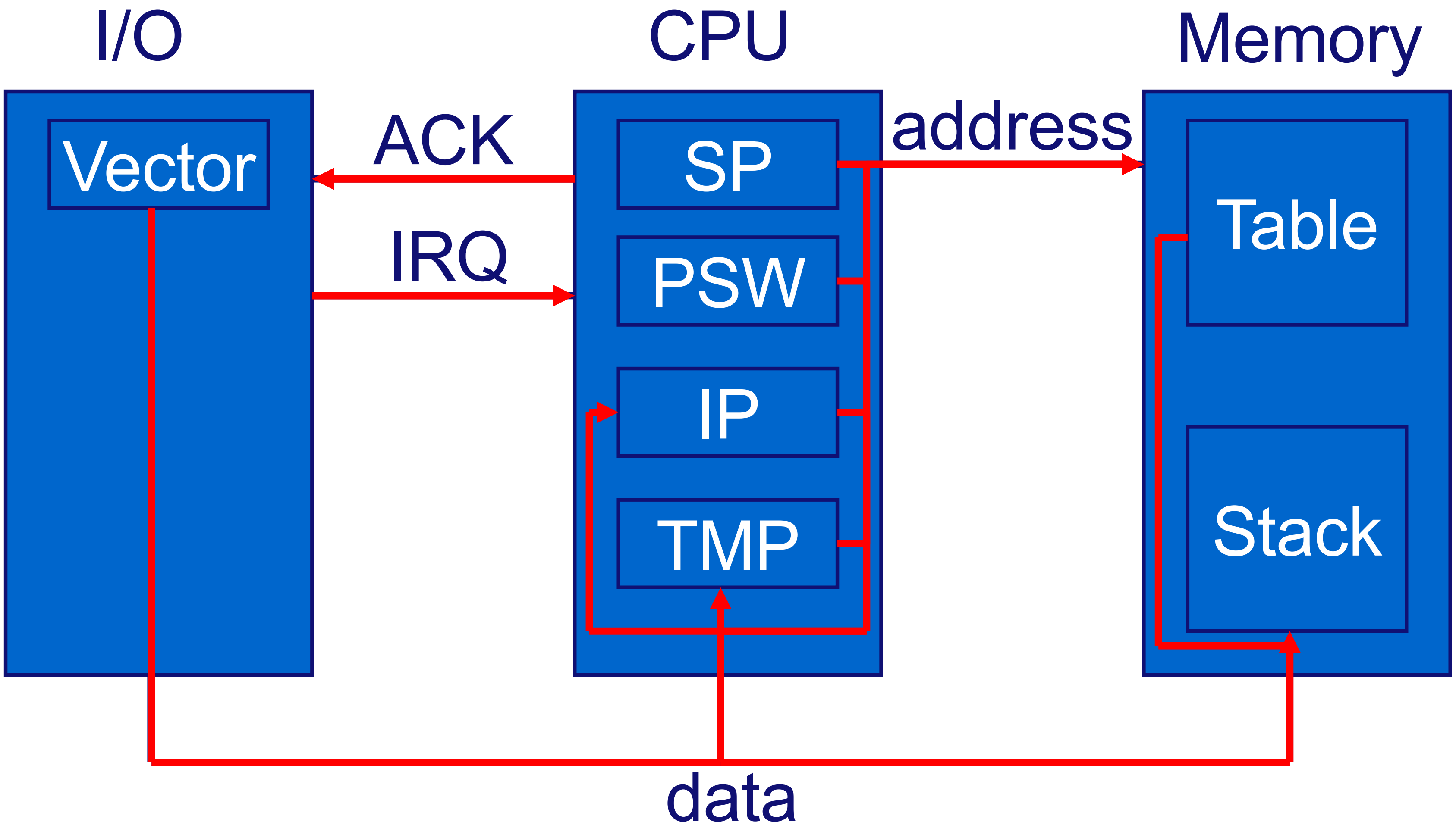
- Call a special subroutine (in supervisor mode on the ARM).
- The effect of a software interrupt is:

  LR :=PC-4   (points to the next instruction)
  PC := 8   (at address 8 there is a jump to the required software handling code)
  CPSR :=APSR   (save the status flags).

Hardware interrupts happen

- Device passes interrupt to CPU
- CPU finishes current instruction and sends an acknowledgement to the device
- Device provides additional information
  (e.g. IRQ-number, target address or table-index)
- {CPU fetches information and temporarily stores this}
- CPU pushes IP (and often PSW) onto the stack
- CPU calls interrupt handler
  (which one depends on the additional information)

Hardware: interrupt (example)



Software: handle interrupt

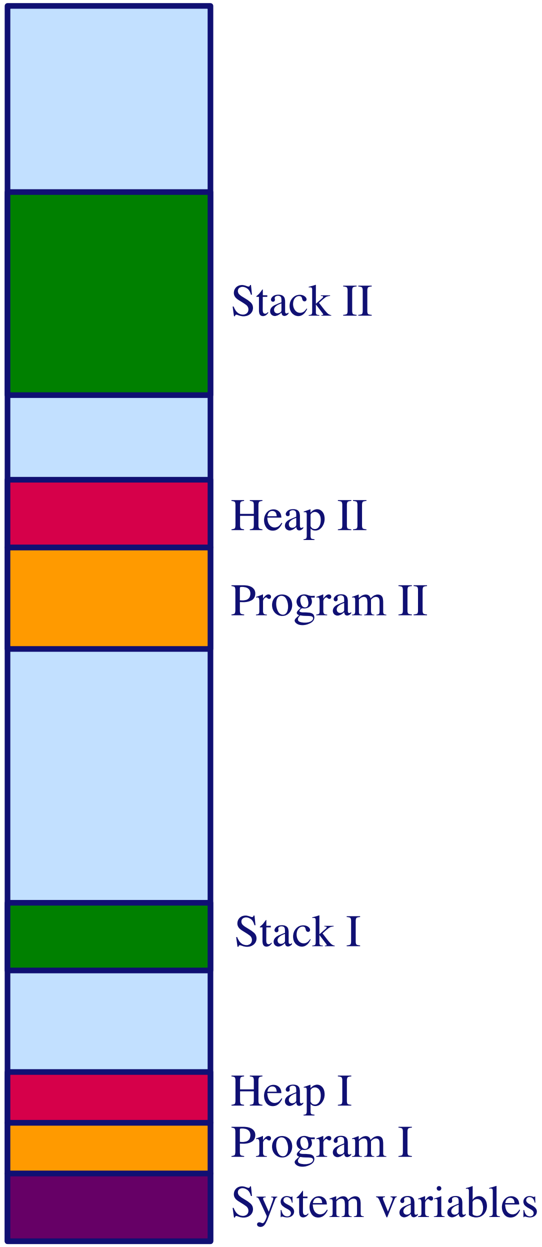
- Save all registers.
- Find out which device caused an interrupt
- Get additional information (status etc).
- Handle the interrupt
  (e.g. I/O error, start next program etc.)
- Signal to the device that the interrupt was serviced
- Restore all registers
- Execute ReTurn from Interrupt instruction, or ERET.

Interrupts: remarks

- Interrupt handlers must be transparent.
- If execution of an interrupt handler takes too long (longer than the period of the interrupt) the main program will not proceed.
- High priority interrupts are used for time critical applications (e.g. while burning a DVD, this was once modern... 😁)
- Interrupts are needed for multitasking.

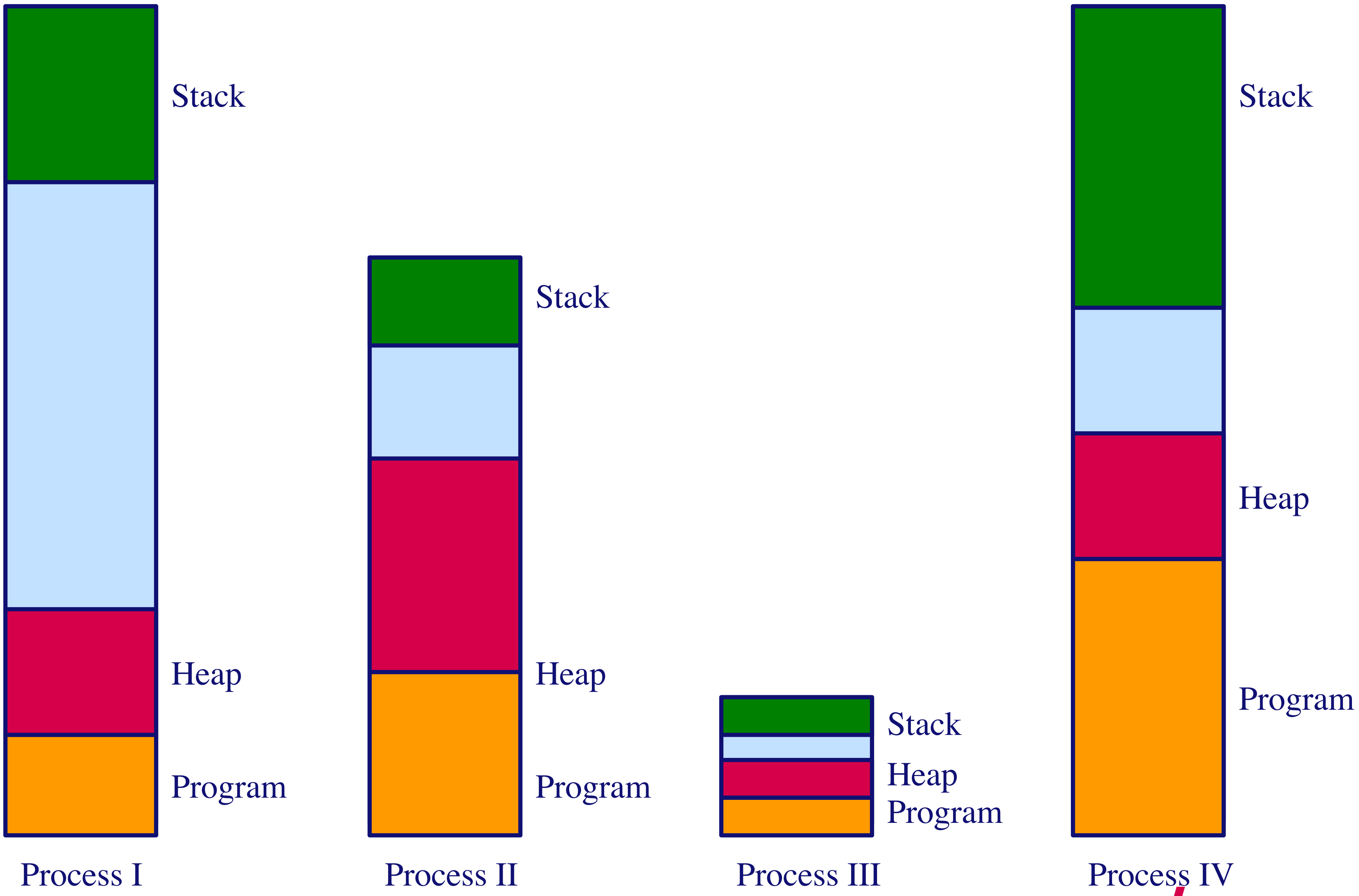
Questions?

Elementary multitasking.



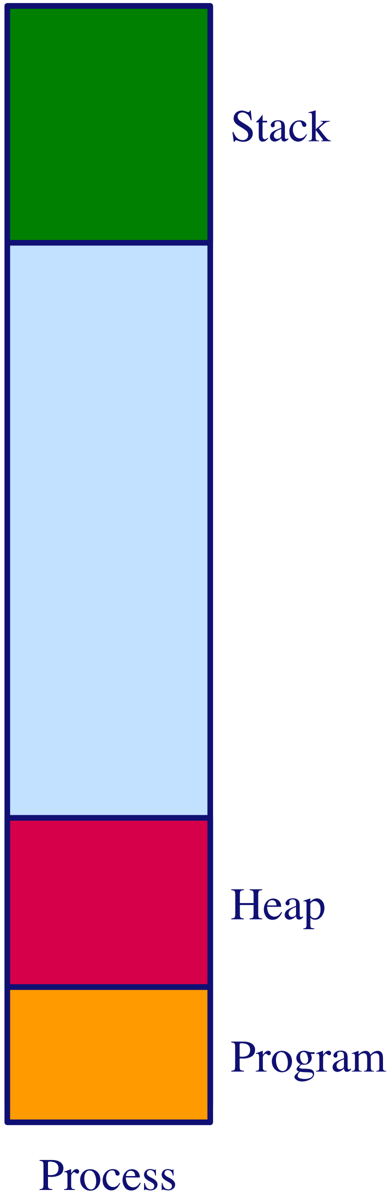
We can store more programs in memory, with their own stack and heap.

Another view on processes.

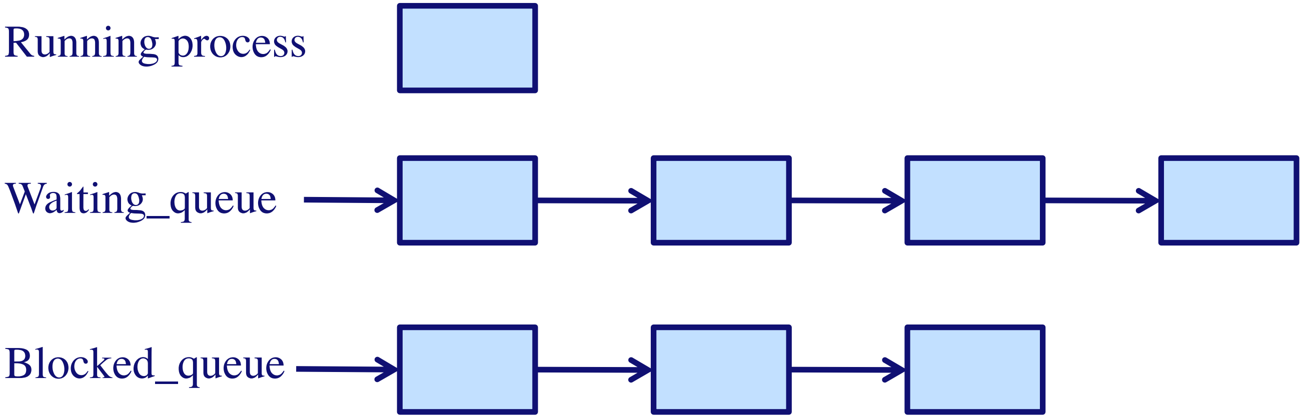


Technische Universiteit
Eindhoven
University of Technology

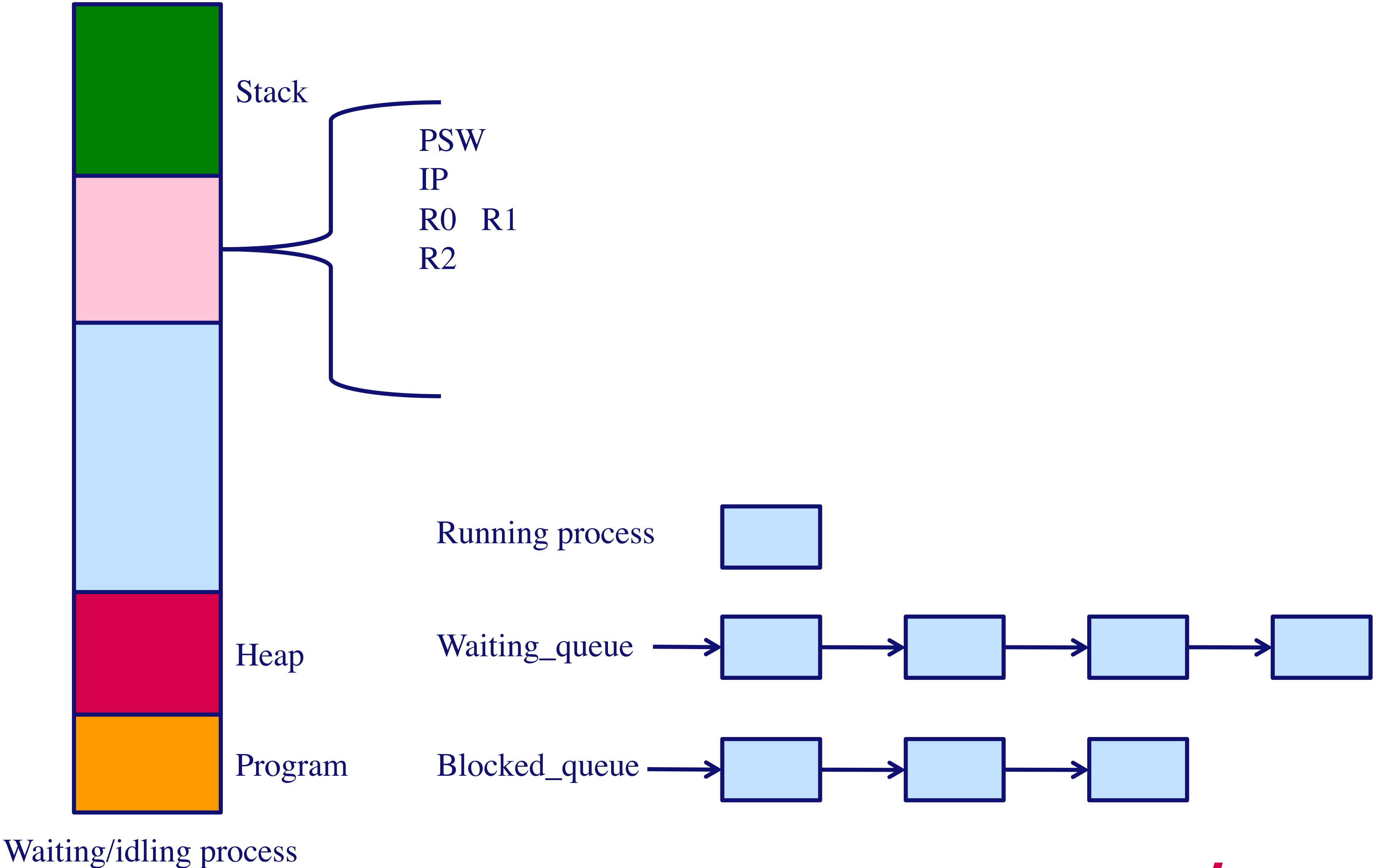
Each process has a process record




```
process_status (running, blocked, waiting,..)

stack_pointer

information_about_heap
information_about_access_rights
information_about_used_objects
etc. etc.
```

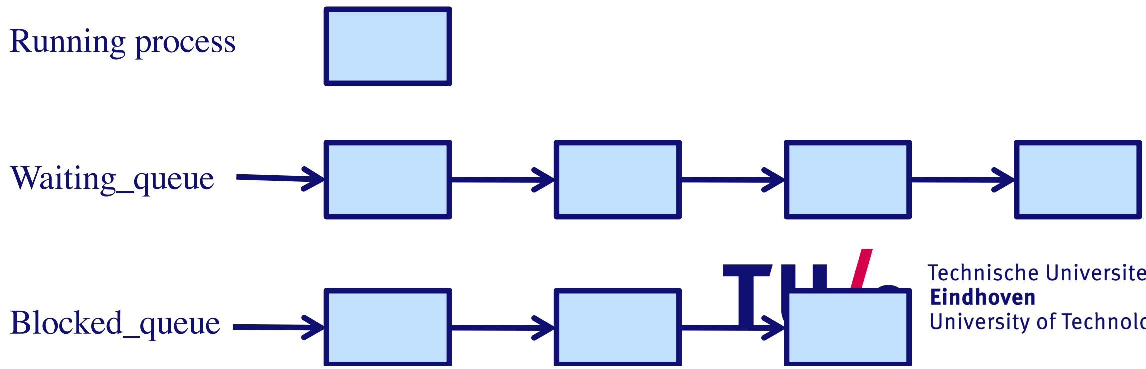


Blocked/waiting processes have info on the stack



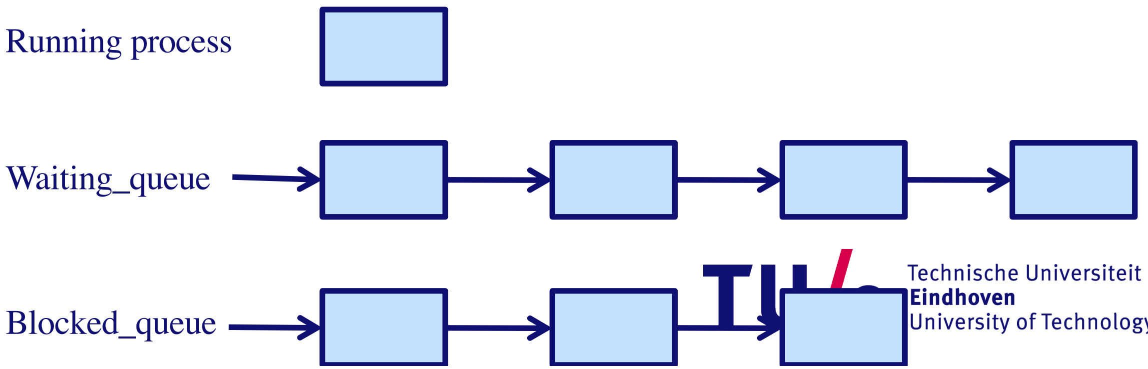
Context switch

```
// When a process timer interrupt comes.

Save registers on stack of the current process;

If the waiting queue is not empty
{
 Save the stackpointer in the current process record;
 Set the status of the current process to wait;
 Put the process record at the end of the waiting list;
 Take a new process record from the waiting queue;
 Set its status to running;
 Put the stackpointer of the new process in the stack register.
}

Restore the registers of the new process;
RTI;  Return from interrupt.
```



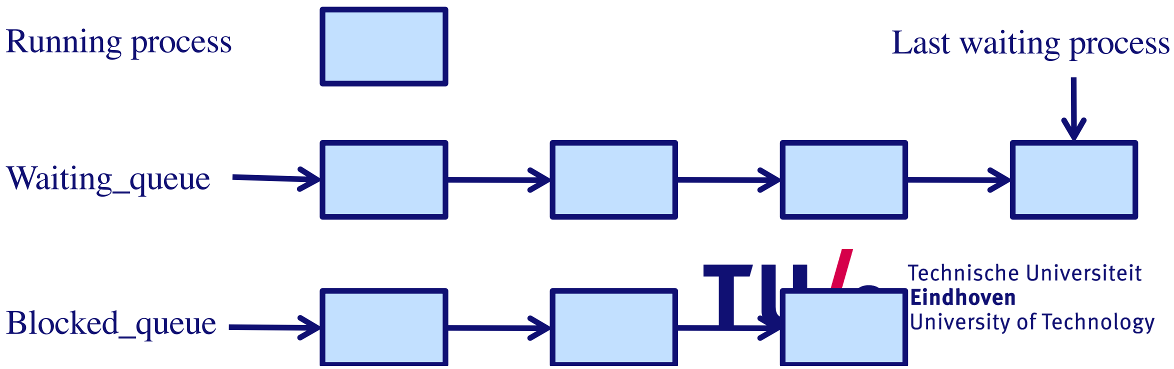
Context switch

```
// When a process timer interrupt comes.

; Save registers on stack of the current process
PUSH R0;

; If the waiting queue is not empty
LOAD R0 [waiting_queue];
BEQ  finish_context_switch;

PUSH R1;
PUSH R2;

; Save the stackpointer in the current process record;
LOAD R1 [running_process]
STOR SP [R1 + stackpointer];

; Set the status of current process to waiting;
LOAD R2 waiting_status;
STOR R2 [R1 + status];
```



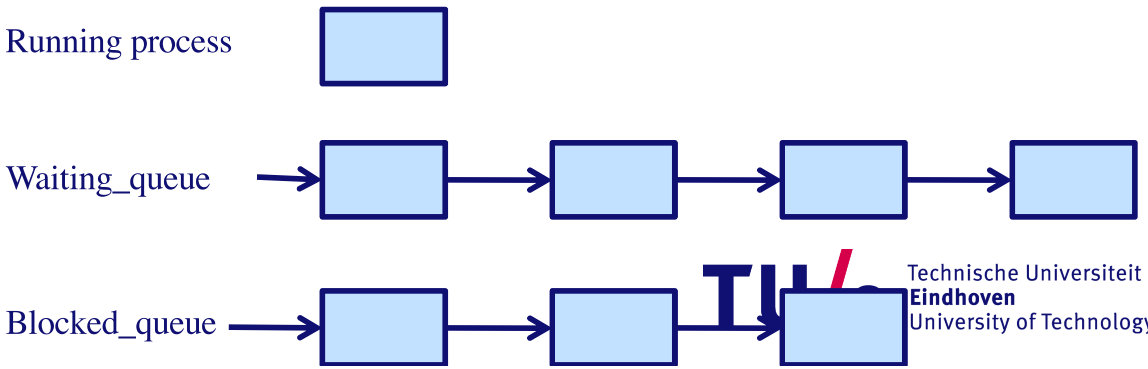
Context switch

```
; Put the process record at the end of the waiting list;

    LOAD R2 [last_waiting_process];
    STOR R1 [R2+next];
    STOR R1 [last_waiting_process];

; Remove the first process record from the waiting queue;
    .....

; Set its status to running;
    LOAD R1 running_status;
    STOR R1 [R0 + status];      R0 points to the new process.

; Put the stackpointer of the new process in the stack register.
    LOAD SP [R0 + stackpointer]
```

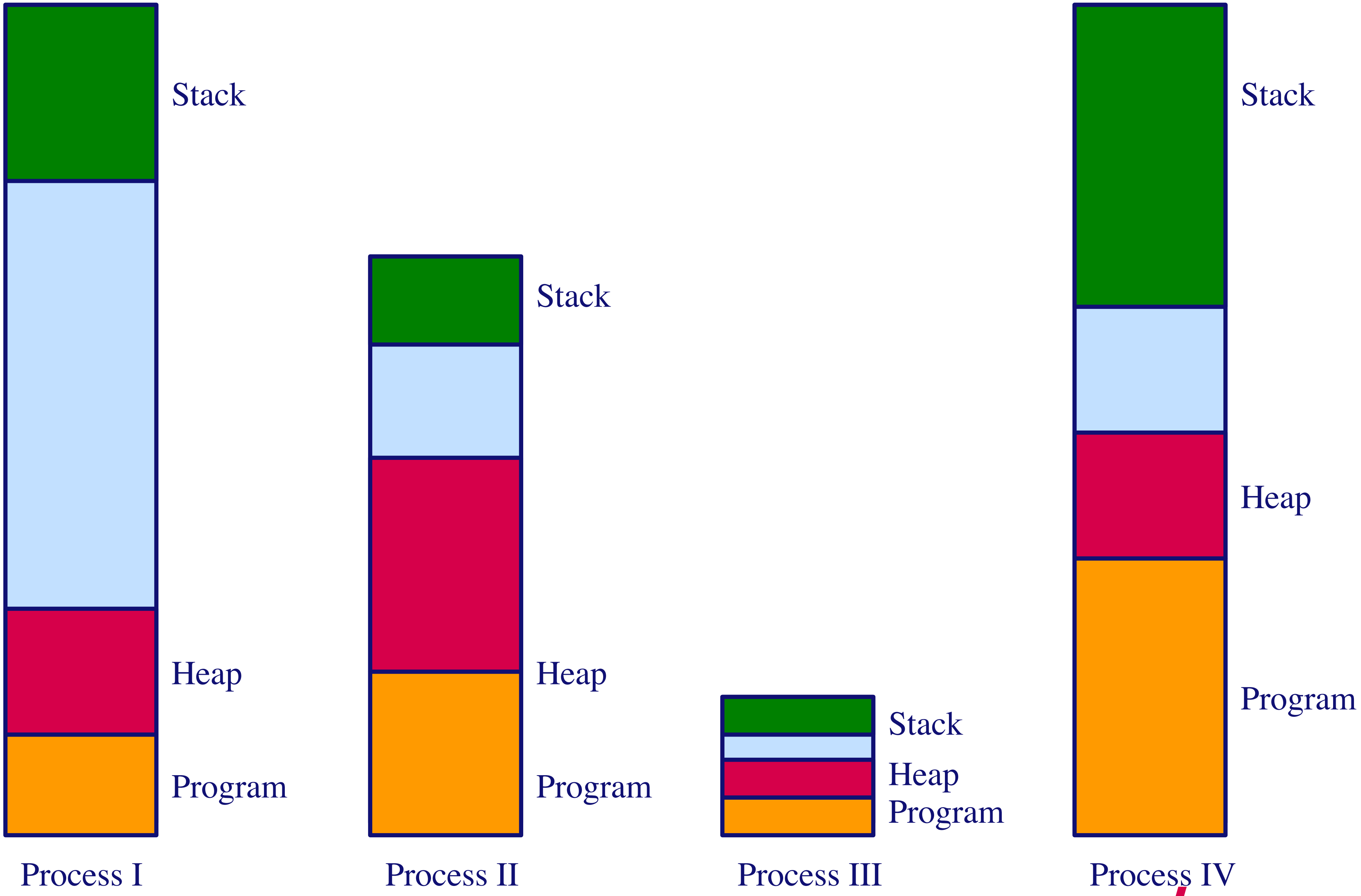


Context switch

```
    PULL R2;
    PULL R1;

finish_context_switch:
; Restore the registers of the new process;
    PULL R0;

    RTI;  Return from interrupt.
```



The processor jumps from process to process.



Questions?

Non atomicity.

Translation to assembly code:

Process 1: $x:=x+1$

Process 2: $x:=x+1$

```
LOAD R0  [BP+x]
ADD R0 1
STOR R0 [BP+x]
```

$\boxed{x=1}$

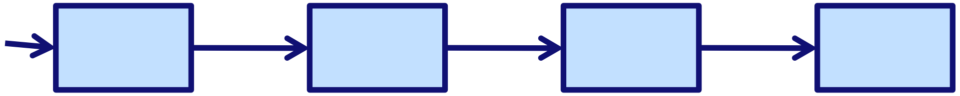
Executing both processes can result in

$\boxed{x=2}$

when both processes are executed simultaneously.
Probability is low. This is hard to test.

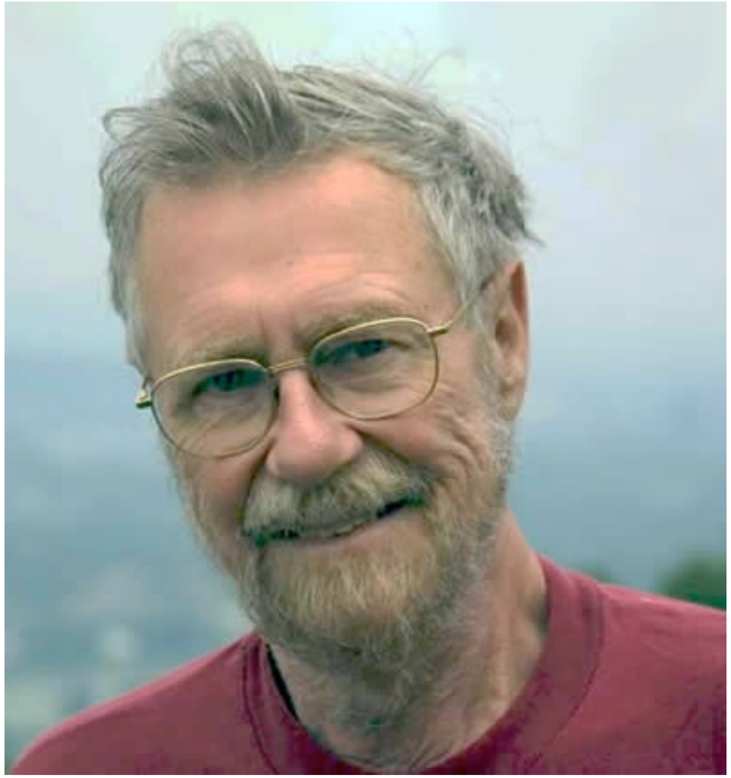
Semaphores/mutexes: tool for process synchronisation

Queue



Passeer (Pass):
- if passable
- then indicate non passability
- else
  - move to blocking queue
  - queue for this semaphore

Vrijgeven (Release):
- if queue is empty
- then indicate passable
- otherwise move a process in the queue
- to the processor waiting queue.



Edsger Wiebe Dijkstra (1930-2002)

These days called mutex variables.

Questions?

Summary

What did we learn:

- We can write programs in assemby (PP2, ARM).
- The stack is used for parameters of functions, local variables, the computation of expressions and returning results.
- We can systematically translate higher level programs into assembly code. This is what compilers do.
- We saw how to transform a single processor machine into a multithreading machine.
- We saw the need for mutual exclusion, and know that semaphores/mutexes can be used for this purpose.