OPERATING SYSTEM
CONTACT SESSION 15

Prof. C R Sarma
WILP.BITS-Pilani

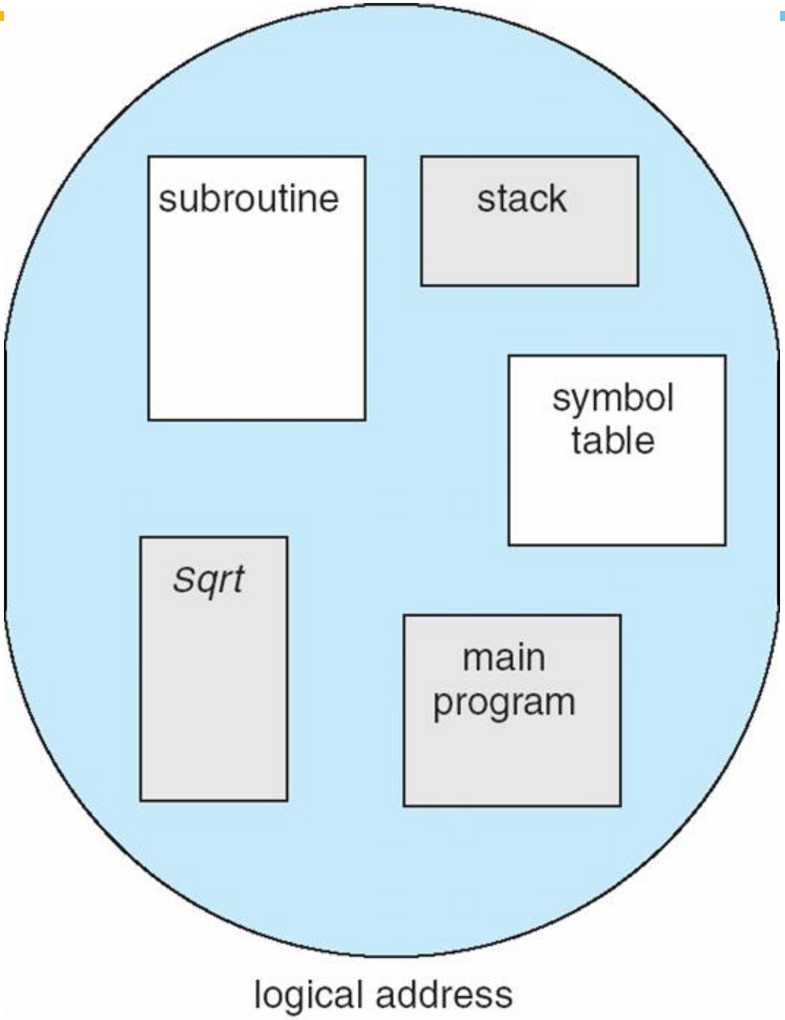
BITS Pilani
Pilani Campus

Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments
  - A segment is a logical unit such as:

    main program
    procedure
    function
    method
    object
    local variables, global variables
    common block
    stack
    symbol table
    arrays
- Two Types : Virtual memory segmentation and Simple Segmentation

User's View of a Program



User specifies each address by two quantities
(a) Segment name
(b) Segment offset

Logical address contains the tuple
<segment#, offset>

Segmentation Architecture

Logical address consists of a two tuple:

‹segment-number, offset›

Segment table – maps two-dimensional logical address to physical address;

Each table entry has:

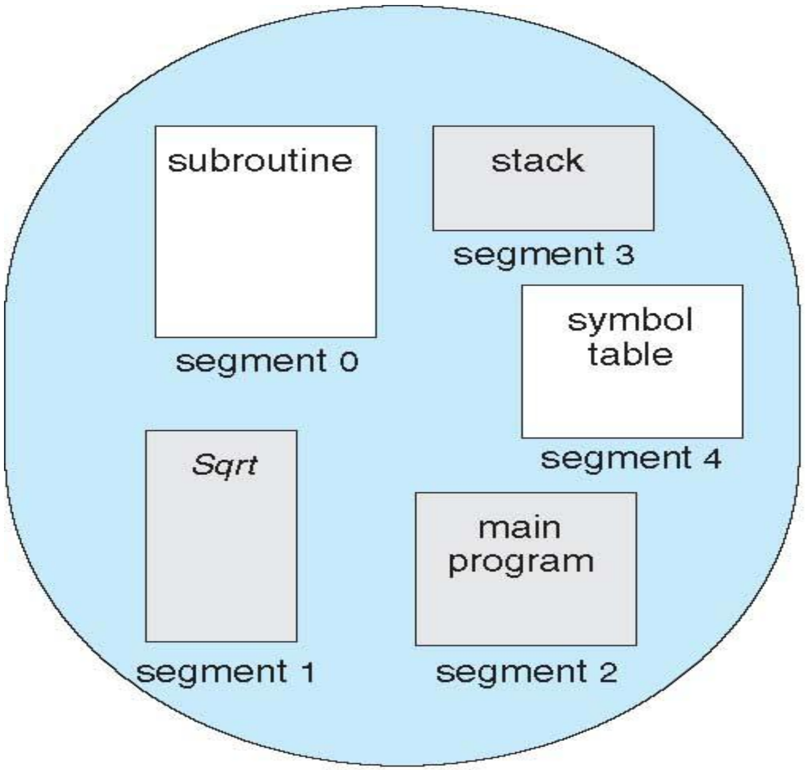
- **base** – contains the starting physical address where the segments reside in memory
- **limit** – specifies the length of the segment

Segment-table base register (STBR) points to the segment table's location in memory

Segment-table length register (STLR) indicates number of segments used by a program;

segment number ***s*** is legal if ***s*** < **STLR**

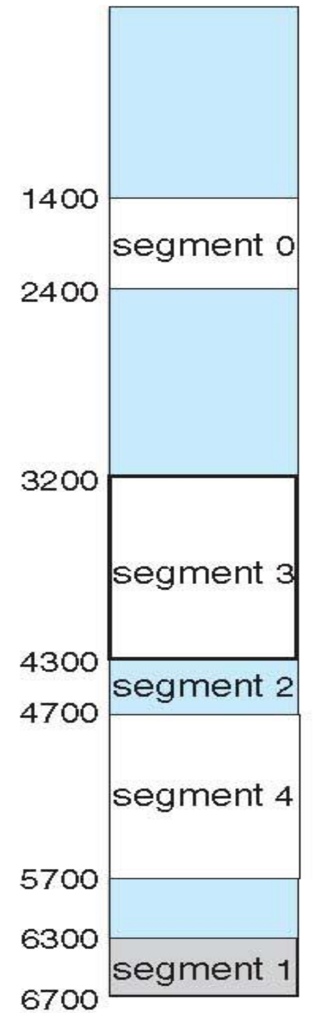
Example of Segmentation



logical address space

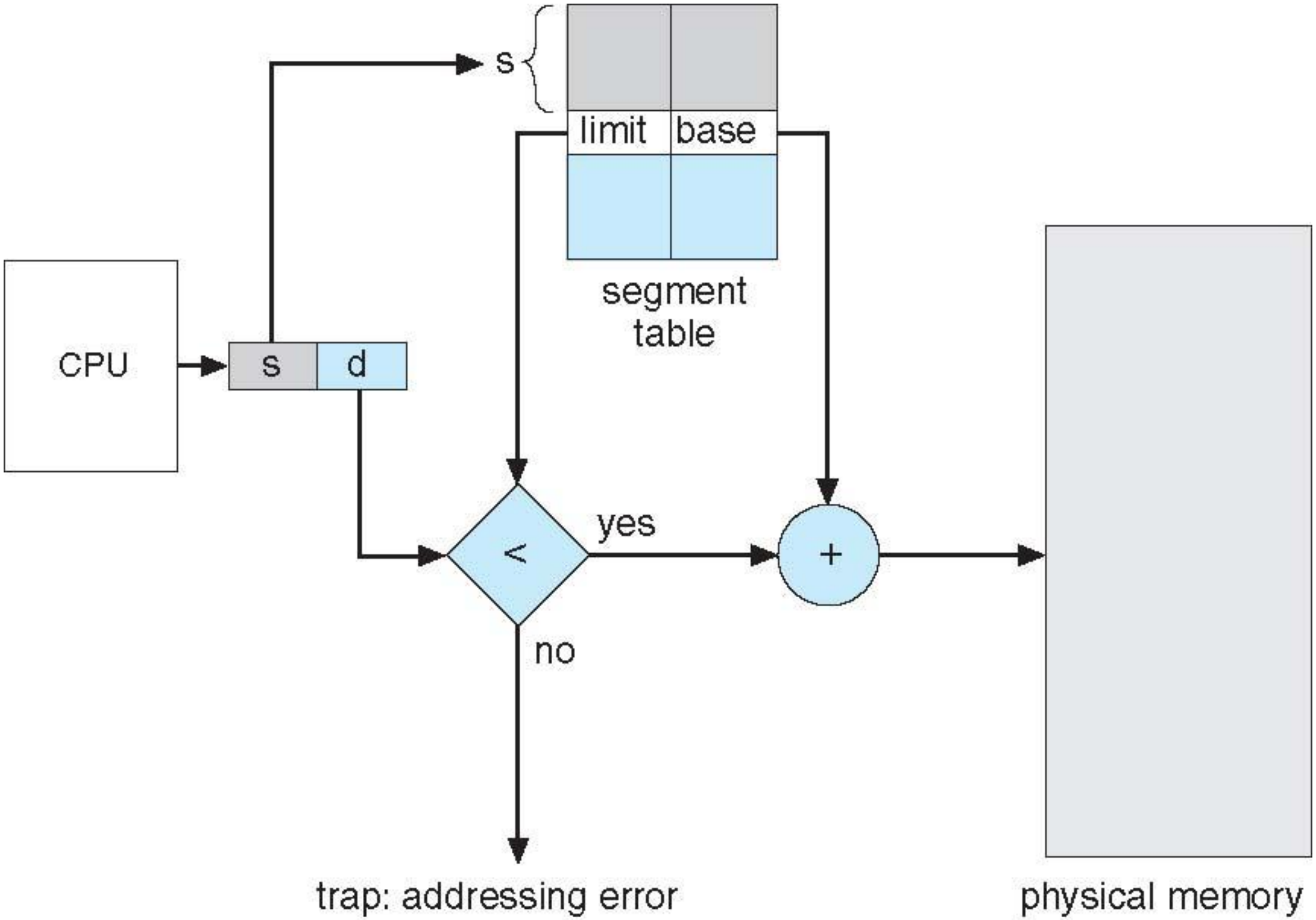
	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table

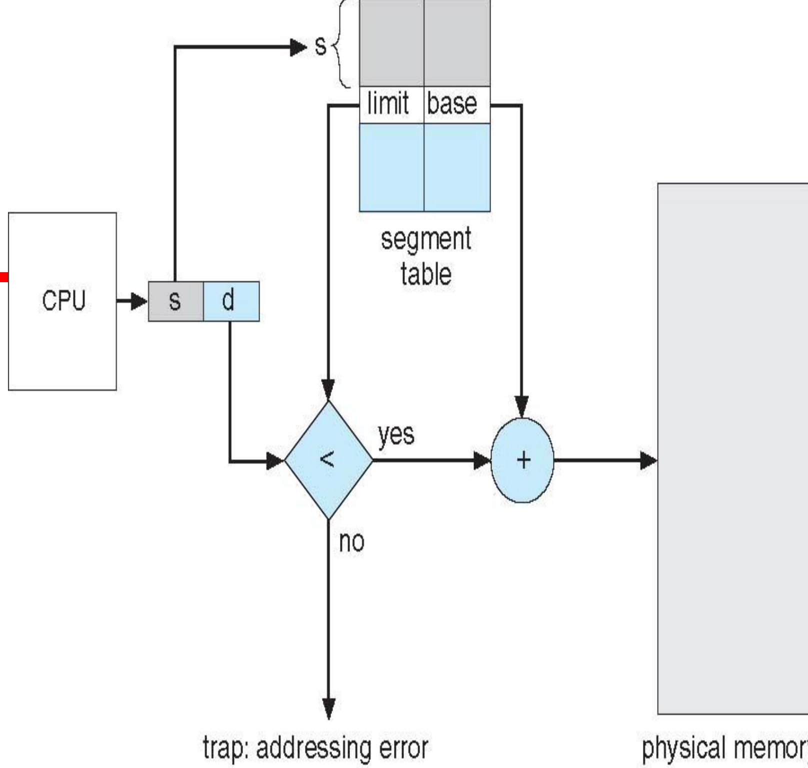


physical memory

Segmentation Hardware



Problem 3: Segmentation



Consider the following segment table:

Segment	Base	Length(Limit)
0	128	512
1	8192	2048
2	1024	4096
3	16384	8192
4	32768	1024
5	65536	16384

What are the physical addresses for the following logical addresses (s, d)?

a) 0, 430

b) 1, 2056

c) 2, 5024

d) 3, 7024

BITS Pilani
Pilani Campus

Optimizing Program Performance

Introduction

- What are the key characteristics of good programming?
- Five elements of a program
  - Variables
  - Loops
  - Conditionals
  - Input/output
  - Functions / methods
- How to write an efficient program?
  - Select an appropriate set of algorithms and data structures
  - Divide the given task and execute the subtasks parallel.
  - Use optimized compiler

Optimizing Compilers

- What is optimizing compiler?
- Steps in optimizing a program
  - Eliminate unnecessary work
  - Instruction level parallelism

Capabilities and Limitations of Optimizing Compilers

```
void twiddle1(int *xp, int *yp)
{
    *xp += *yp;
    *xp += *yp;
}
```

```
void twiddle2(int *xp, int *yp)
{
    *xp += 2* *yp;
}
```

```
void twiddle1(int *xp )
{
      *xp += *xp;
      *xp += *xp;
}

void twiddle2(int *xp )
{
     *xp += 2* *xp;
}
```

Code Optimization blocker – Memory Aliasing

```
x = 1000; y = 3000;
*q = y;     /* 3000 */
*p = x;     /* 1000 */
t1 = *q;    /* 1000 or 3000 */
```

```
x = 1000; y = 3000;
*p = x;     /* 1000 */
t1 = *q;    /* 1000 */
```

Example 1

Consider the following procedure to swap two values:

```
/* Swap value x at xp with value y at yp */
void swap(int *xp, int *yp)
{
    *xp = *xp + *yp; /* x+y       */
    *yp = *xp - *yp; /* x+y-y = x */
    *xp = *xp - *yp; /* x+y-x = y */
}
```

If this procedure is called with xp equal to yp, what effect will it have?

Code Optimization blocker – Function Calls

```
int f();

int func1() {
    return f() + f() + f() + f();
}
```

```
int func2() {
    return 4*f();
}
```

```
int counter = 0;
int f() {
    return counter++;
}
```

Optimization techniques

- Code Movement
- Dead Code Elimination
- Strength Reduction
- Common Expression Elimination
- Compile time evaluation
  - Constant Folding
  - Constant Propagation

Code Movement

Move the code fragment outside the loop as it won't have any difference if it is performed inside the loop repeatedly or outside the loop once.

Source Code:

```
for ( x = 0 ; x < n ; x++ ) {
    temp = sum + 10;
    a[x] = a[x] + x;
}
```

Optimized Code:

```
temp = sum + 10;
for ( x = 0 ; x < n ; x++ ) {
    a[x] = a[x] + x;
}
```

Contd…

```
void set_row(double *a, double *b,
   long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

→

```
    long j;
    int ni = n*i;
    for (j = 0; j < n; j++)
        a[ni+j] = b[j];
```

Dead Code Elimination

Remove **code** fragment which does not affect the program results.

```
int fun(void) {
  int x = 4;
  int y = 5; /* Assignment to dead variable */
  int z;
  z = x  +  4;
  return z;
  x = 4; /* Unreachable code */
  return 0;
}
```

```
int fun(void) {
  int x = 4;
  int z;
  z = x  +  4;
  return z;
}
```

Strength Reduction

Replace complex instructions with cheaper expressions.

Source Code :

```
for ( x = 0; x < n ; x++){
  y = z * 2;
}
```

Optimized Code :

```
for ( x = 0; x < n ; x++){
  y = z + z;
}
```

Common Expression Elimination

- Eliminate the expression which is appearing repeatedly in the code

```
Source code:
x1 =  y * 2 + z;
x2 = y * 2 - z;
```

```
Temp = y * 2;
x1 =  Temp + z;
x2 = Temp - z;
```

Compile Time Evaluation

- Constant Folding: Process of evaluating the expressions whose values are constant at compile time.

Example :

Source code :
```
x = y + 2 + z - 3;
```

Optimized code :
```
x = y + z - 1;
```

- Constant Propagation: Process of substituting the values of known constants in the expressions at compile time

Source Code :
```
int x = 10;
int y = x + 10 + x/2 ;
return y + x;
```

Optimized Code 1:
```
int x = 10;
int y = 10 + 10 + 10 /2;
return y + 10;
```

Optimized Code 2:
```
int x = 10;
int y = 25;
return 35;
```

Loop Unrolling

- Also known as loop unwinding
- Tries to transform the loop so that program execution improves at the cost of size

Source Code:

```
while ( x <= 100) {
    a[x] = x+10;
    x++;
}
```

Optimized Code :

```
while ( x <= 100) {
    a[x] = x+10;
    x++;
    a[x] = x+10;
    x++;
}
```

Example

```
int x;
for (x = 0; x < 100; x++)
{
    delete(x);
}
```

```
int x;
for (x = 0; x < 100; x += 5 )
{
    delete(x);
    delete(x + 1);
    delete(x + 2);
    delete(x + 3);
    delete(x + 4);
}
```

BITS Pilani
Pilani Campus

Practical Approach

Experiment 1: Redundant Code

```
program Ex1
    n = 5
    for i = 1 to 6
        n = n + 1
        if n = 3 then
            n = 0
        end if
    next
end
```

b.In the COMPILER frame click on the Enable "Optimizer" check box. Click on the "Redundant Code" check box in the Optimizer window. Compile the source again.

Observe the following:

i. Note down the code size in Binary Code -> Show button -> Show instructional Stats

ii. Note the lean and mean set of instructions in the assembly code generated

Experiment 2: Constant Folding

```
program Ex4
    n = 1 + 7 - 9
end
```

Repeat what was done in experiment 1 and notice the changes.

a. Next check the "Constant Folding" check box.

Experiment : 3
Strength Reduction

```
program Ex5
    i = 3
    n = i * 16
end
```

Repeat what was done in experiment 1 and notice the changes.

a. Next check the Strength Reduction check box

Experiment 4: Loop Unrolling

```
program Ex6
    for p = 1 to 30
        r = r + 2
    next
end
```

Repeat what was done in experiment 1 and notice the changes.

a. Next check the Loop Unrolling check box

Compile time is more, Code size is large compared to un-optimized code

b. Execute the program with and without optimiser and see the execution time difference innovate achieve lead

That's all for penultimate session CS# 15 Back together on Finale session CS#16