Bab 14

Activation Records dan Stack Management

Sub-CPMK yang Dicakup dalam Bab Ini:

- **Sub-CPMK 5.3:** Mengimplementasikan activation records untuk procedure calls

14.1 Pengenalan Activation Records

14.1.1 Definisi Activation Record

Activation Record (stack frame) adalah struktur data yang menyimpan:

- Local variables
- Parameters
- Return address
- Saved registers
- Dynamic link (pointer ke caller frame)
- Static link (pointer ke enclosing scope)

14.1.2 Stack Frame Layout

```
// Typical stack frame layout (grows downward)
+--------------------+  // Higher addresses
| Parameters         |  (from caller)
+--------------------+
| Return Address     |  (pushed by CALL)
+--------------------+
```

```
| Dynamic Link       |  (old frame pointer)
+--------------------+
| Saved Registers    |  (caller-saved)
+--------------------+
| Local Variables    |  (allocated by callee)
+--------------------+
| Temporaries        |  (temporary storage)
+--------------------+  // Lower addresses
```

14.2 Procedure Call Mechanism

14.2.1 Calling Convention

Prosedur call untuk function `func(a, b, c)`:

```
// Caller side:
push c              // Push parameters (right-to-left)
push b
push a
call func           // Push return address and jump

// Callee side (func entry):
push ebp            // Save old frame pointer
mov ebp, esp         // Set new frame pointer
sub esp, local_size // Allocate space for locals

// Function body here...

// Callee side (func exit):
mov esp, ebp         // Deallocate locals
pop ebp             // Restore old frame pointer
ret                 // Pop return address and jump
```

14.2.2 Parameter Passing

Method	Pros	Cons
Stack	Simple, unlimited parameters	Slow, memory access
Register	Fast, no memory access	Limited registers
Mixed	Balance of speed	Complex

Tabel 14.1: Parameter Passing Methods

14.3 Stack Management

14.3.1 Stack Operations

```c
typedef struct {
    void *base;         // Stack base
    void *top;          // Stack top
    size_t size;        // Current size
    size_t capacity;    // Maximum size
} Stack;

void push(Stack *stack, void *data, size_t data_size) {
    if (stack->size + data_size > stack->capacity) {
        stack_overflow_error();
    }

    memcpy(stack->top, data, data_size);
    stack->top = (char*)stack->top + data_size;
    stack->size += data_size;
}

void* pop(Stack *stack, size_t data_size) {
    if (stack->size < data_size) {
        stack_underflow_error();
    }

    stack->top = (char*)stack->top - data_size;
    stack->size -= data_size;
    return stack->top;
}
```

14.3.2 Frame Pointer vs Stack Pointer

```c
// Using frame pointer (EBP/RBP)
int access_local(int offset) {
    // Access local at [EBP - offset]
    return *(int*)((char*)EBP - offset);
}

// Using stack pointer only (ESP/RSP)
int access_local_no_fp(int offset_from_sp) {
    // Access local at [ESP + offset]
    return *(int*)((char*)ESP + offset_from_sp);
}
```

14.4 Nested Functions

14.4.1 Static Links

Untuk nested functions (Pascal-style):

```c
typedef struct {
    void *return_addr;
    void *dynamic_link;  // Pointer to caller frame
    void *static_link;   // Pointer to enclosing function frame
```

```
    LocalVars locals;
    Temporaries temps;
} ActivationRecord;

void nested_function() {
    // Access variable from enclosing function
    ActivationRecord *current = get_current_frame();
    ActivationRecord *enclosing = current->static_link;

    int outer_var = enclosing->locals.some_variable;
}
```

14.4.2 Display Method

Alternative untuk nested functions:

```
typedef struct {
    ActivationRecord *frames[MAX_NESTING];
    int current_level;
} Display;

void access_nested_var(int level, int offset) {
    ActivationRecord *frame = display.frames[level];
    return *(int*)((char*)frame + offset);
}
```

14.5 Exception Handling

14.5.1 Stack Unwinding

```
typedef struct {
    void *handler;
    void *frame_pointer;
    int handler_type;
} ExceptionHandler;

void throw_exception(int exception_type) {
    // Unwind stack looking for handler
    ActivationRecord *current = get_current_frame();

    while (current != NULL) {
        ExceptionHandler *handler = find_handler(current, exception_type)
↪ ;
        if (handler) {
            // Jump to handler
            longjmp(handler->jump_buffer, exception_type);
        }
        current = current->dynamic_link;
    }

    // No handler found - terminate
```

```
    terminate_program();
}
```

14.6 Optimization Techniques

14.6.1 Leaf Function Optimization

```
// Before optimization
int leaf_function(int x, int y) {
    return x + y;
}

// After optimization (no frame setup)
int leaf_function_optimized(int x, int y) {
    // Use registers only, no stack operations
    return x + y;
}
```

14.6.2 Tail Call Optimization

```
// Before optimization
int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);  // Need to preserve frame
}

// After tail call optimization
int factorial_tail(int n, int acc) {
    if (n <= 1) return acc;
    return factorial_tail(n - 1, n * acc);  // Can reuse frame
}
```

14.6.3 Register Saving Optimization

```
// Save only registers that are actually used
void save_used_registers(RegisterMask used_regs) {
    for (int i = 0; i < NUM_REGISTERS; i++) {
        if (used_regs & (1 << i)) {
            push_register(i);
        }
    }
}

void restore_used_registers(RegisterMask used_regs) {
    for (int i = NUM_REGISTERS - 1; i >= 0; i--) {
        if (used_regs & (1 << i)) {
            pop_register(i);
        }
    }
```

```
}
```

14.7 Variable Length Arrays

14.7.1 VLA on Stack

```
void function_with_vla(int n) {
    int vla[n];  // Variable length array

    // Stack layout after VLA allocation:
    // +--------------------+
    // | VLA (n * sizeof(int)) |
    // +--------------------+
    // | Other locals       |
    // +--------------------+
    // | Frame pointer      |
    // +--------------------+

    // Access VLA
    for (int i = 0; i < n; i++) {
        vla[i] = i * 2;
    }
}
```

14.7.2 Alloca Implementation

```
void* alloca(size_t size) {
    void *result = (char*)current_frame_pointer - size;

    // Check for stack overflow
    if (result < stack_limit) {
        stack_overflow_error();
    }

    // Adjust stack pointer
    current_stack_pointer = result;
    return result;
}
```

Aktivitas Pembelajaran

1. **Stack Frame**: Implementasikan stack frame management system.

2. **Calling Convention**: Bangun procedure call mechanism dengan berbagai conventions.

3. **Nested Functions**: Implementasikan static links untuk nested functions.

4. **Exception Handling**: Buat exception handling dengan stack unwinding.

5. **Optimization**: Implementasikan leaf function dan tail call optimizations.

Latihan dan Refleksi

1. Gambarkan stack frame layout untuk function dengan multiple parameters dan locals!

2. Implementasikan calling convention untuk variadic functions!

3. Analisis overhead dari frame pointer vs stack pointer only!

4. Implementasikan exception handling dengan proper cleanup!

5. Optimasi function calls dengan tail recursion elimination!

6. **Refleksi**: Bagaimana activation records mempengaruhi performance function calls?

Asesmen (Evaluasi Kinerja)

Instrumen Penilaian untuk Sub-CPMK 5.3

A. Pilihan Ganda

1. Dynamic link menunjuk ke:
   (a) Enclosing function frame
   (b) Caller frame
   (c) Global frame
   (d) Stack base

2. Tail call optimization menghilangkan:
   (a) Parameter passing
   (b) Frame setup overhead
   (c) Return value
   (d) Function call

3. Static link digunakan untuk:

(a) Exception handling

(b) Nested functions

(c) Recursion

(d) Optimization

B. Essay

1. Jelaskan complete procedure call mechanism dengan activation records!
2. Implementasikan stack management system dengan support untuk nested functions dan exception handling!

Rubrik Penilaian: Lihat Lampiran A

Checklist Pencapaian Kompetensi

Centang item berikut setelah Anda yakin telah menguasainya:

☐ Saya dapat mengimplementasikan activation records untuk procedure calls

☐ Saya dapat mengelola stack operations dan frame management

☐ Saya dapat mengimplementasikan berbagai calling conventions

☐ Saya dapat menangani nested functions dengan static links

☐ Saya dapat mengimplementasikan exception handling

☐ Saya dapat melakukan stack-related optimizations

Rangkuman

Bab ini membahas activation records dan stack management, termasuk stack frame layout, procedure call mechanisms, nested functions, exception handling, and optimization techniques. Mahasiswa belajar mengimplementasikan efficient function call infrastructure.

Poin Kunci:

- Activation records menyimpan state untuk function execution
- Stack management mengelola dynamic allocation/deallocation
- Calling conventions menentukan parameter passing dan register saving
- Static links enable access to enclosing function variables

- Exception handling requires proper stack unwinding
- Optimizations reduce overhead of function calls

Kata Kunci: ***Activation Record***, ***Stack Frame***, ***Calling Convention***, ***Dynamic Link***, ***Static Link***, ***Tail Call Optimization***, ***Exception Handling***