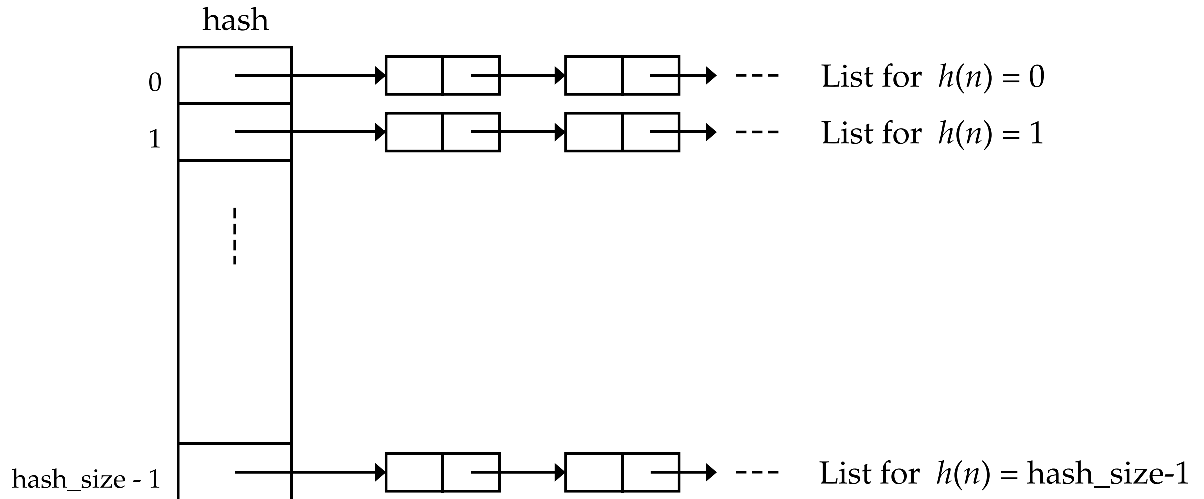

Figure 2.1: A hashing scheme

⟨Symbol table global variables 11a⟩ ≡

```
#define hash_size 353  /* should be prime */
names hash[hash_size]; /* heads of hash lists */
```

Macro defined by scraps 11a, 13g, 14c, 18bd, 21b.
Macro referenced in scrap 23d.

Initially all the hash lists are empty. Although strictly speaking the array `hash` should be initialised to null pointers by the compiler, we don't count on this.

⟨Initialise symbol table globals 11b⟩ ≡

```
{ int i=hash_size; do hash[--i]=NULL; while(i>0); }
```

Macro defined by scraps 11b, 13h, 14d, 18ce, 21c.
Macro referenced in scrap 23d.

Here is the main function for finding names in the hash table. The parameter `ident` points to a null-terminated string.

⟨Symbol table prototypes 11c⟩ ≡

```
extern name find_name (char *ident);
```

Macro defined by scraps 11c, 13e, 14bf, 15a, 16ae, 17ac, 20ac, 21ad, 22a.
Macro referenced in scrap 23a.

⟨Symbol table functions 11d⟩ ≡

```
name find_name (char *ident)
{ int h;
  ⟨Compute the hash code h of the string ident 12a⟩
  ⟨Find and return the associated name, possibly by entering a new identifier into the table 12b⟩
}
```

Macro defined by scraps 11d, 14g, 15b, 16bf, 17bd, 20bd, 21e, 22b.
Macro referenced in scrap 23d.

A simple hash code is used: If the sequence of character codes is $c_1c_2\ldots c_n$, its hash value will be

$$(2^{n-1}c_1 + 2^{n-2}c_2 + \cdots + c_n) \bmod \texttt{hash_size}.$$

name is returned as result. Note that if the object represented by the name occupies more than one stack slot, e.g. it is an array, then the offset returned will be to the beginning of the object (see Figure 2.3).

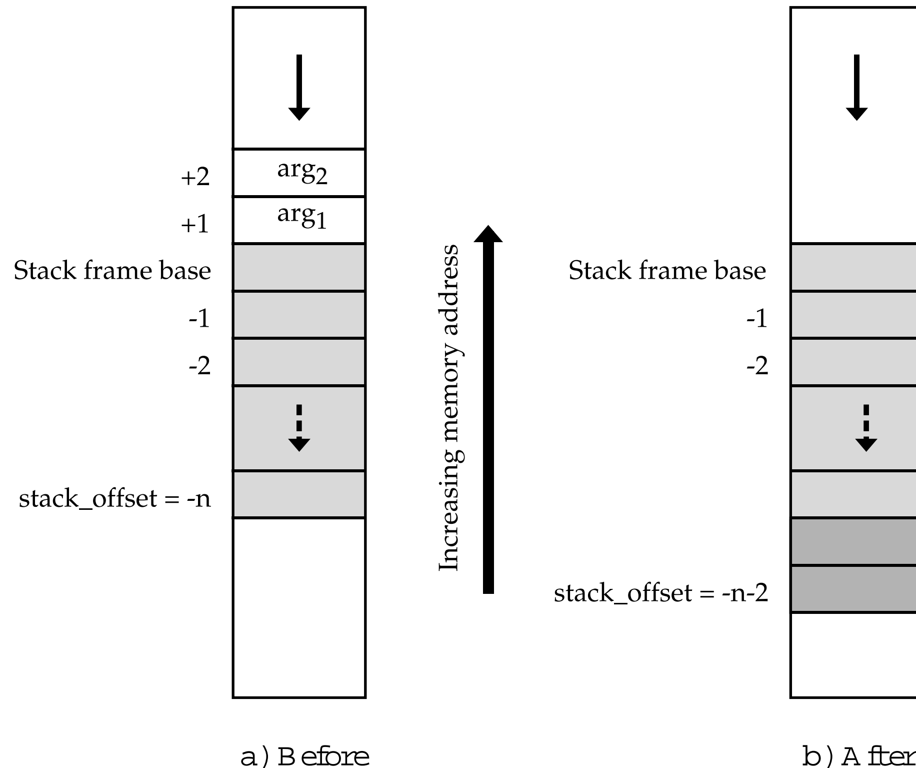

Figure 2.2: Effect of the call `insert_name(..., 2, ...)`

⟨Symbol table prototypes 15a⟩ ≡

```
extern int insert_name(name name, int size, type type);
```

Macro defined by scraps 11c, 13e, 14bf, 15a, 16ae, 17ac, 20ac, 21ad, 22a.
Macro referenced in scrap 23a.

⟨Symbol table functions 15b⟩ ≡

```
int insert_name(name name, int size, type type)
{ symbol_table_entry s;
  if (stack_offset > 1) stack_offset = 1; /* In case we have inserted args */
  if (type != VAL_T) stack_offset -= size;
  insert_name_with_offset(name, stack_offset, size, type);
  return stack_offset;
}
```

Macro defined by scraps 11d, 14g, 15b, 16bf, 17bd, 20bd, 21e, 22b.
Macro referenced in scrap 23d.

Arguments to procedures will be accessed with positive offsets from the base of the stack frame (see Section 5.2) and so we provide a variant of the `insert_name` function that adds `size` to `stack_offset` at each call. In this case we know that `type` is restricted by the language to `INT_T` or `CHAN_T`, and hence `size` is always 1.

Chapter 5

The μOCCAM Abstract Machine

The μOCCAM abstract machine is basically a simple stack machine extended with instructions to support processes and communication. We start by describing the sequential aspects of the machine and then deal with the complications introduced by concurrency. The machine implementation errs on the side of clarity rather than efficiency.

Note: The machine doesn't make many safety checks. It is up to the code generator to ensure that stacks don't overflow, jumps are in range etc.

5.1 The Sequential Machine

The machine uses a stack to hold the program variables and intermediate values. The code for the program is stored in a separate array. Two machine registers, `SP` and `PC` are used to index into these components, as illustrated in Figure 5.1.

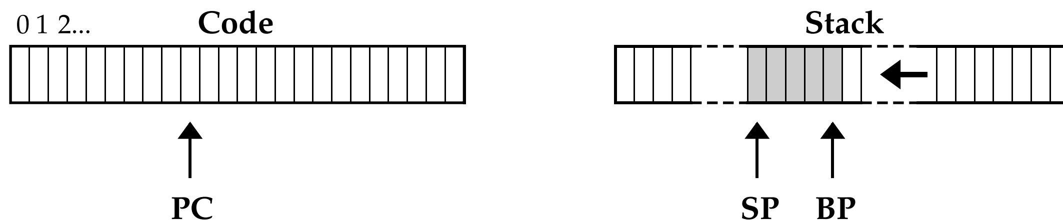

Figure 5.1: The sequential machine state

An instruction consists of two components. The `op` field determines the instruction type. The `arg` field contains an argument for the operation. Only some of the operations take arguments. The value in the `arg` field is ignored in the other cases. We assume that integers occupy thirty-two bits. It turns out to be convenient if instructions can occupy the same amount of space as ints, and so we restrict the size of the argument field to twenty-four bits. This should be more than adequate for our needs. Table 5.1 on page 33 lists the complete instruction set of the abstract machine.

⟨Machine opcodes `typedef` 31⟩ ≡

```
typedef enum opcode
{ ⟨Machine opcodes 32e, ... ⟩ } opcode;
```

Macro referenced in scrap 77e.

⟨Stack effect of executing instruction `curr_inst` 42c⟩ ≡

```
case Call : return(1);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

To return we need to pop the stack, if necessary, until the return address is on top, and then execute the `Ret` instruction.

⟨Machine opcodes 42d⟩ ≡

```
Ret,
```

Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 42e⟩ ≡

```
case Ret : { PC = (SP++)->as_int; break; }
```

Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

The stack effect might seem a bit odd. However, the effect is from the view of the procedure currently executing, not the procedure being returned to.

⟨Stack effect of executing instruction `curr_inst` 42f⟩ ≡

```
case Ret : return(0);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

Arguments to a procedure should be placed on the stack *before* the `Enter` instruction is evaluated. They can then be accessed by the called procedure using positive offsets from `BP`. Thus a typical calling sequence looks like

Evaluate m arguments

Enter n

Call L /* Call procedure at nesting depth n */

Leave n

Pop m

The stack during the execution of the procedure is illustrated in Figure 5.2.

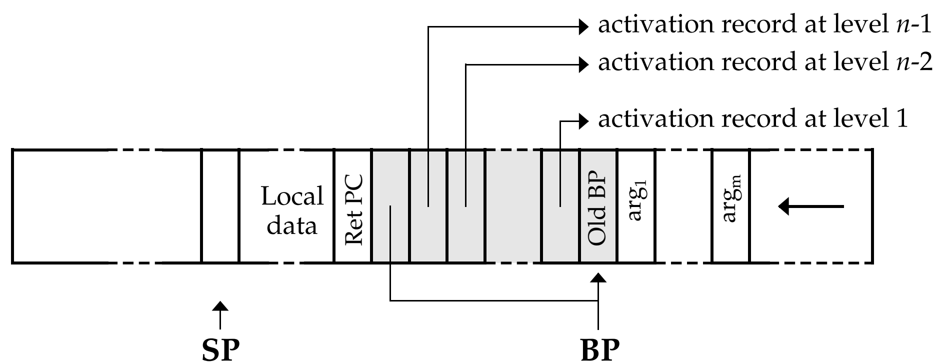

Figure 5.2: A typical activation record at nesting level n

5.3 The Parallel Machine

A process may spawn many subprocesses. Each of these must be allocated its own stack, otherwise chaos reigns. However, each subprocess must be able to access (but not update) variables in its parent's stack if they are in scope. We must therefore provide a mechanism for accessing the stacks of our ancestors. We can use the display mechanism for this purpose. We treat the spawning of a process as a degenerate kind of procedure call. The display at the beginning of the new stack contains a pointer to the activation record of the spawner. However, unlike a procedure call, we don't need to save the PC as processes don't return to the caller.

The state of the abstract machine, from the viewpoint of a single process, now looks like Figure 5.3.

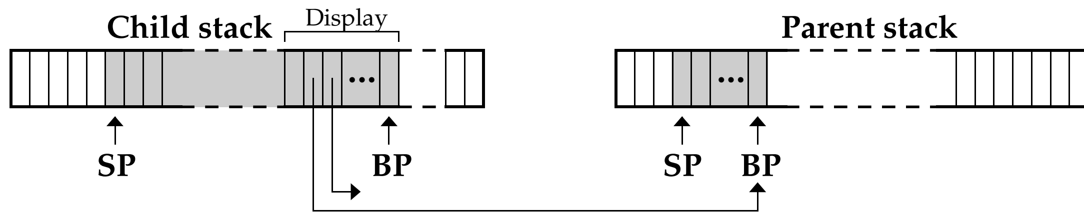

Figure 5.3: The parallel machine state

In order to create a process we must specify the stack size to be used for the new process, and the initial value of its program counter. Note that all processes share the same code array. The compiler needs to calculate the maximum stack size required by the process (including the display at the base of the stack). The language is constrained enough that the maximum can be calculated at compile time. The `Spawn` instruction can be used to spawn a new process. This creates the data structures required by the new process and schedules it for execution. The process counter is supplied as an argument to the instruction and the stack size is passed on the top of the stack. The nesting level of the new process is supplied as the next element on the stack.

⟨Machine opcodes 43a⟩ ≡

```
Spawn,
```

Macro defined by scraps 32e, 34adh, 35cf, 36ad, 37adg, 38be, 39adg, 40c, 41ae, 42ad, 43a, 44be, 45ad, 46adg, 47d, 48c, 49c, 61a.
Macro referenced in scrap 31.

⟨Execute instruction `curr_inst` 43b⟩ ≡

```
case Spawn : {
  process p = create_process(curr_inst.arg, (SP++)->as_int);
  p->BP = BP;
  initialise_display((SP++)->as_int, &p->SP, &p->BP);
  ⟨Schedule process p for execution 71c⟩
  break; }
```

Macro defined by scraps 32f, 34bf, 35adg, 36be, 37beh, 38cf, 39be, 40ad, 41bf, 42be, 43b, 44cf, 45be, 46be, 47be, 48d.
Macro referenced in scrap 71b.

⟨Stack effect of executing instruction `curr_inst` 44a⟩ ≡

```
case Spawn : return(-2);
```

Macro defined by scraps 32g, 34cg, 35beh, 36cf, 37cf, 38adg, 39cf, 40be, 41dg, 42cf, 44adg, 45cf, 46cf, 47cf, 48e, 49d.
Macro referenced in scrap 61c.

Sometimes it is useful to be able to pass an initial value to the spawned process (e.g. in the case of a replicated `PAR`). The `Spawnv` instruction allows you to do this.

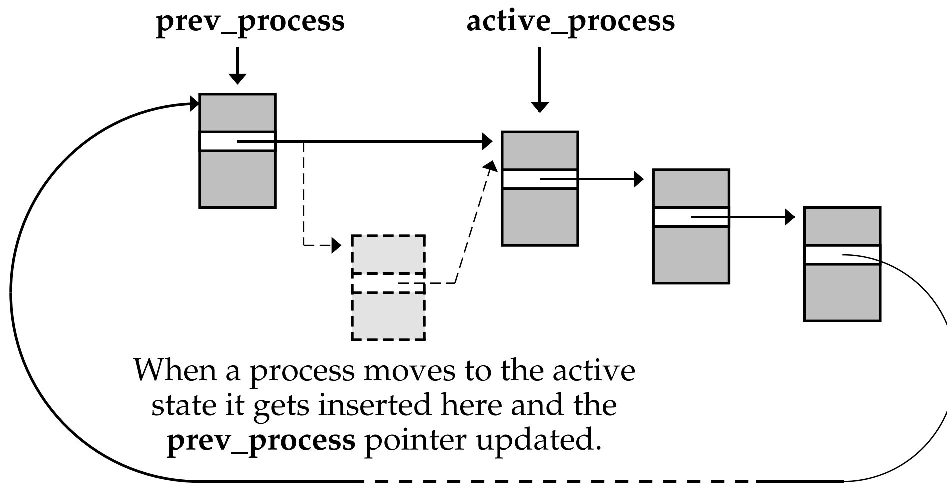

Figure 5.4: The active process queue

```
process create_process(int PC, int stacksize)
{ process p;
  ⟨Allocate storage for process p with a stack of size stacksize 53f⟩
  p->PC = PC;
  p->next = NULL;
  p->children = 0;
  p->parent = active_process;
  if (active_process) active_process->children++;
  p->stack_size = stacksize;
  p->status = ACTIVE;
  p->SP = (stack_slot *)((char *)p + sizeof(process_block)) + stacksize;
  p->BP = NULL;
  p->stack_overflow_check = OVERFLOW_CHECKSUM;
  p->step_process = p->trace_process = FALSE;
  return p;
}
```

Macro defined by scraps 41c, 50b, 51e, 54b, 58ab, 59ac, 60ab, 61c, 62d, 63acd, 66c, 67a, 69c, 73d.
Macro referenced in scrap 78c.

When a process dies by executing a `Stop` instruction we must decide whether to retire it or reincarnate it straight away. This requires knowing how many spawned children are still alive. We therefore add a `children` field to the PCB to keep track of this.

⟨Other PCB fields 52a⟩ ≡

```
int children;
```

Macro defined by scraps 48b, 51b, 52a, 77a.
Macro referenced in scrap 50e.

⟨Terminate current process 52b⟩ ≡

```
{ process p = active_process;
  ⟨Remove p from active process queue 53a⟩
  if (p->children) { /* Retire as there are still children alive */
    p->status = RETIRED; }
  else ⟨Reincarnate process p 53c⟩
}
```

Macro referenced in scraps 44f, 55b, 56a.

When we remove a process from the active process queue we must be careful to deal with the cases where there is only one process.