The Runtime Stack

CS 350: Computer Organization & Assembler Language Programming

A. Why?

- Information for a procedure call (arguments, caller information, results) is stored in an activation frame.
- At runtime, the activation frames form a stack (in C and C++) or heap (in Java), and this is used to implement procedure calls and returns.

B. Outcomes

After this lecture, you should know

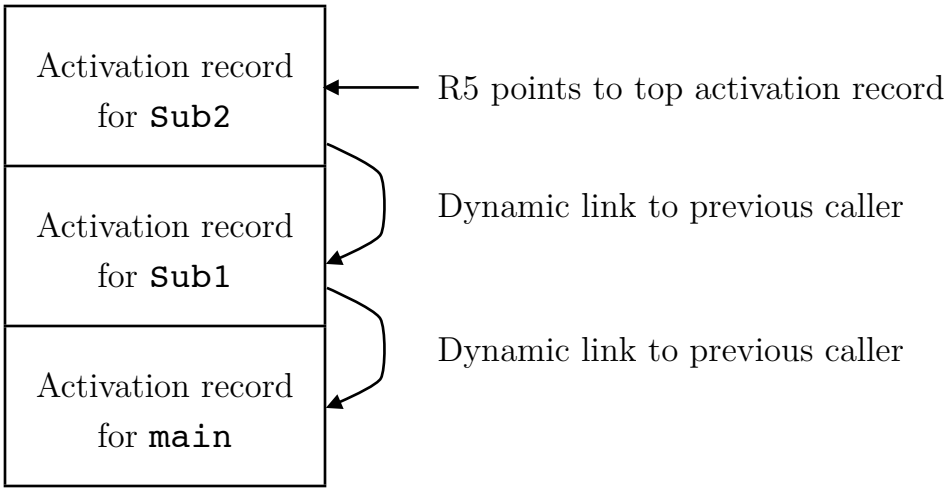
- What activation frames look like and how the runtime stack is used to save procedure call information.

C. Subroutine Calls in C and C++

- Higher-level languages support recursive subroutines, and the simple save/restore technique for variables doesn't work for them.
- In C and C++, the solution is to use the **runtime stack**. The elements of the runtime stack are called **activation records** or **activation frames**; each one holds the information we need to execute a call (what we're passing, what we need returned, where to return to, and the local variables and parameters of the routine we're calling).
- If `main` calls *subroutine 1* calls *subroutine 2*, … calls *subroutine n*, then the runtime stack will contain an entry for each of these routines (`main`, *subroutine 1*, etc.) with `main` being the oldest.
- The topmost frame is for the active call. When we finish with it, we go back to the most-recently paused call; this is why we need a stack (last in, first out).
- When we return from a subroutine call, the memory space for that call is popped off the stack, deallocated, and sent back to the runtime system for re-use. This is why in C, a routine should not return a pointer to a local

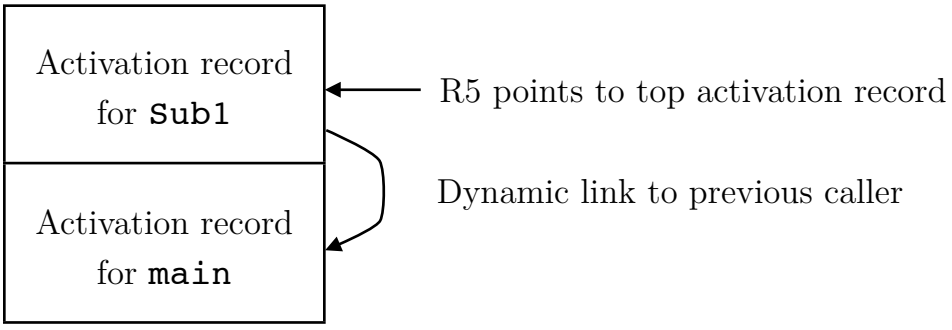
variables. (Java doesn't deallocate the memory space when we return from a call; it relies on a runtime garbage collector to find and deallocate unused space as necessary.)

- The Patt & Patel book uses `R5` to point to the top activation record; each record contains a "dynamic link" field that points to its predecessor record in the stack. For example, if `main` calls `Sub1` calls `Sub2`, the runtime stack looks like this:



Runtime Stack (`main` called `Sub1` called `Sub2`)

- When `Sub2` returns to `Sub1`, we have



Runtime Stack (`main` called `Sub1`

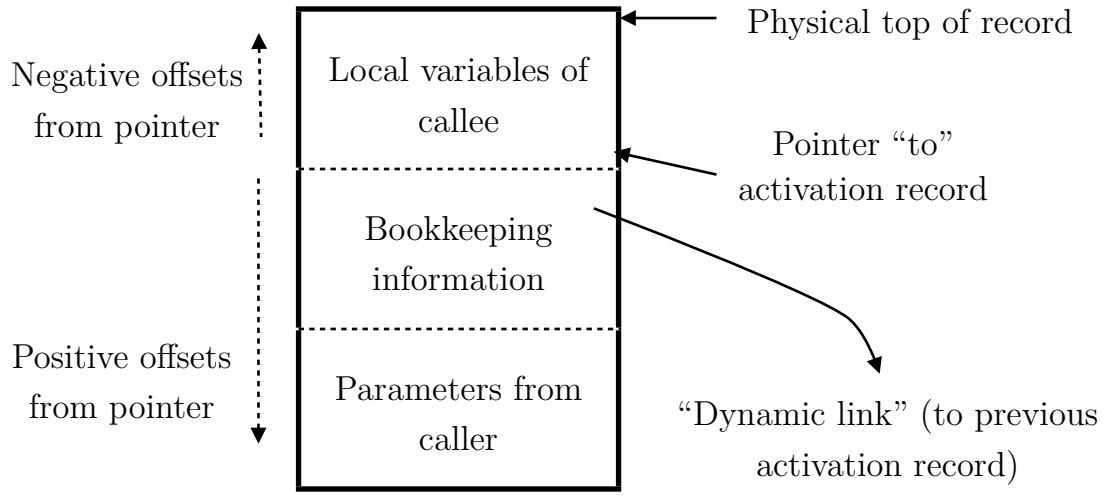
- When `Sub1` returns to main, we have a one-record stack, and when main returns to the operating system, the stack is empty.

D. Contents of Activation Record

- An activation record for a call of a routine contains space for its local

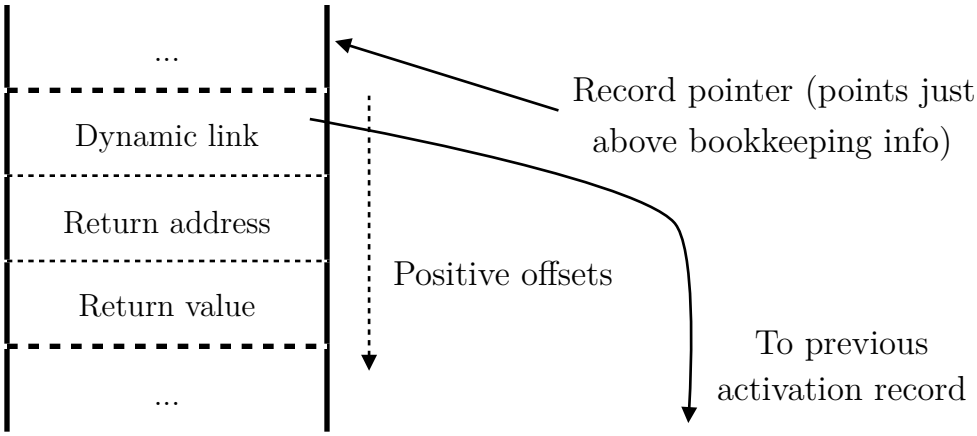
variables, some bookkeeping information, and space for its parameters. Since different routines can have different numbers and sizes of local variables and parameters, different routines can have activation records of different lengths.

- Because different routines have activation records of different lengths, Patt & Patel "point to" an activation record by pointing to the bookkeeping information. (Strictly speaking, to the address before the bookkeeping information.) This way, the bookkeeping fields have the same offsets for all routines.



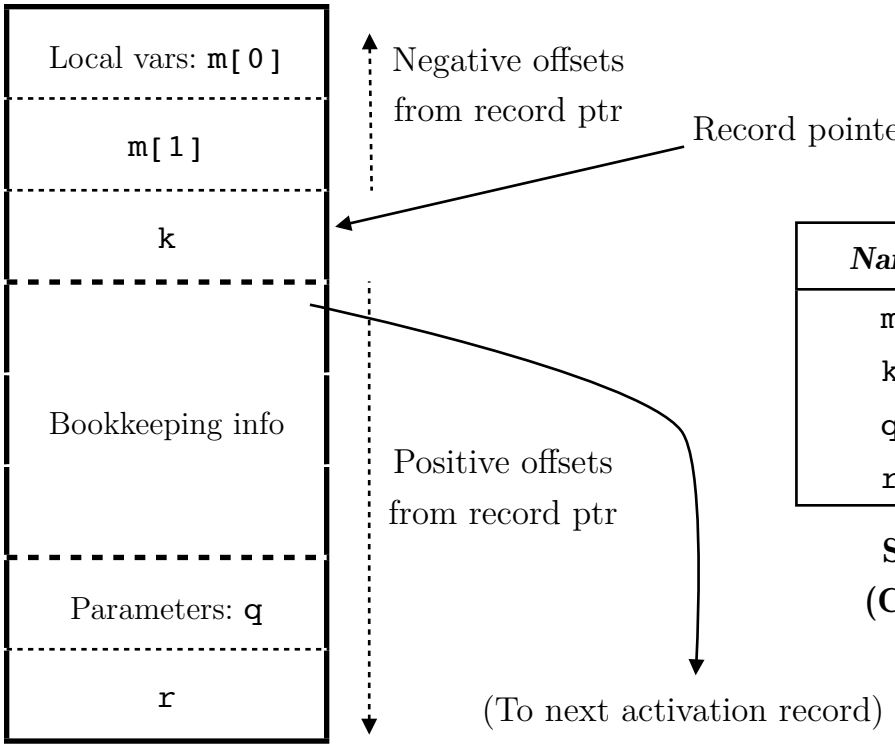
Typical Runtime Activation Record

- The bookkeeping information is always at the same location relative to the pointer "to" the record. It includes
  - The **dynamic link**, which points to the activation record of the caller.
  - The **return address** to which we should go, once the call is completed.
  - Space for the **return value**, which will be filled in by the callee as part of completing the call. (This space may be unused; depends on the routine.)



Bookkeeping Information in Activation Record

- The local variables have negative offsets from the record pointer; the parameters have positive offsets.
- **Example**: for `Sub2(int r, int q) { int k, m[2]) { … }`, we get



Name	*Offset*
m	-2
k	0
q	4
r	5

Symbol Table (Compile-Time)

Activation Record for `Sub2`

E. How to Call a Subroutine

- To push or pop an activation record requires pushes or pops of multiple words of data. The textbook uses `R6` as the pointer to the top of the physical word-oriented stack that's used to implement the runtime stack.
  - To **push** the value in register R to the stack, we use

    ```
    STR R, R6, 0
    ADD R6, R6, -1
    ```

  - To **pop** the top of the stack into a register R, we use

    ```
    ADD R6, R6, 1
    LDR R, R6, 0
    ```

- To call a procedure (say routine P calls Q)
  1. The caller P sets up the parameters of Q by calculating the argument values; then it pushes them (right to left) onto the stack.
  2. Then P does a `JSR`'s to the routine being called (saving `R7` ← `PC` establish the address to return to.
  3. The callee Q sets up the rest of the activation record:
     a. Q pushes space for the return value.
     b. Q sets the return address by pushing `R7`.
     c. Q sets the dynamic link by pushing `R5`.
     d. Q copies `R5` ← `R6` so that the "current" activation record is the one we're building.
     e. Q pushes spaces for its local variables, if any.
     f. Then we jump to the code for the body of Q and start executing it.
- As the called routine Q executes, it can use `LDR`/`STR` *register*`, R5, `N to access the local variables and parameters. (The offset N depends on the variable/parameter.)
- To return from the call of Q back to P,
  1. The routine Q that is returning has to
     a. Pop off any local variables

   b. Pop the dynamic link from stack into `R5`. (This makes the "current" activation record the one for the caller *P*.)
   c. Pop the caller's return address into `R7`.
   d. Copy the return value to the top slot of the stack.
   e. Return (`RET` = `JMP R7`)
2. Then the routine *P* being returned to has to:
   a. Pop off the return value that was pushed on by the called routine.
   b. Pop off and throw away the arguments it pushed on before the call.

F. Sample of Nested Call

- In the example below, the `main` program calls `Sub1` which calls `Sub2`:

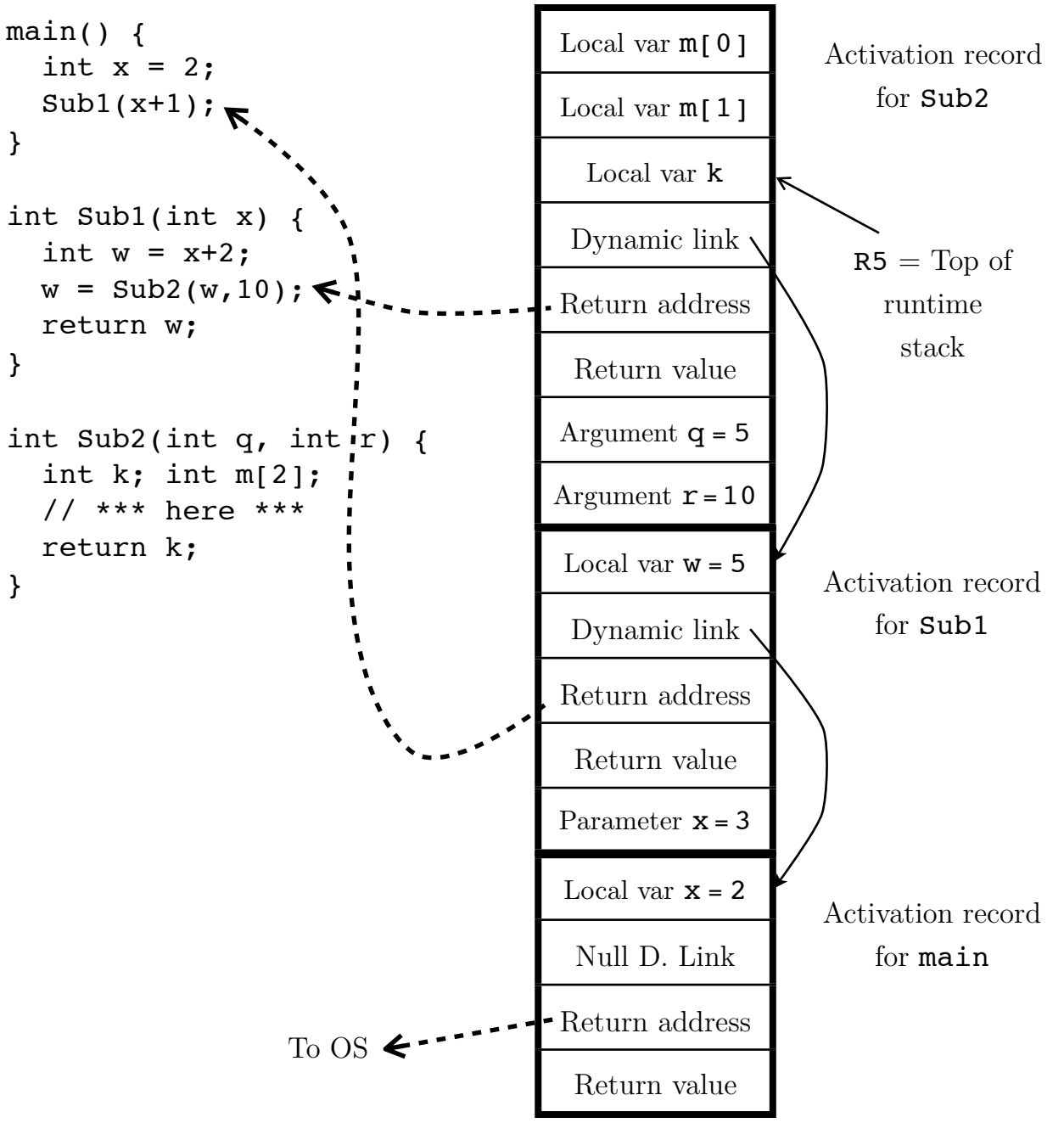
```
main() {
  int x = 2;
  Sub1(x+1);
}


int Sub1(int x) {
  int w = x+2;
  w = Sub2(w,10);
  return w;
}


int Sub2(int q, int r) {
  int k; int m[2];
  // *** Here ***
  return k;
}
```

- Since `main`, `Sub1`, and `Sub2` have 0, 1, and 2 integer parameters respectively and 1, 1, and 3 integers of local space respectively, their activation records are of sizes 4 = 0+3+1, 5 = 1+2+1, and 8 = 2+3+3 respectively.
- Note that since the `x` of the `main` program and the `x` of `Sub1` are in different routines, their values must be stored in different places.

- At the point marked "`Here`" in `Sub2`, the activation stack looks like this:



The Runtime Stack

CS 350: Computer Organization & Assembly Language Programming

A. Why?

- Subroutines are the most basic way to share executable code.
- In operating systems, stacks can help track nested subroutine calls.

B. Outcomes

After this activity, you should be able to

- Sketch the runtime stack at various points during the execution of a program.

C. Questions

1. Suppose in the `Sub1`/`Sub2` example, `Sub2` returns `8`. Sketch the runtime stack after `Sub2` returns to `Sub1` and `Sub1` assigns `w = Sub2(…)`.
2. Continuing from Question 1, sketch the runtime stack just after `Sub1` returns to the `main` program.
3. Sketch code that `Sub2` could use to implement `k = q+r;` Use the symbol table

Name	*Offset*
`m`	`-2`
`k`	`0`
`q`	`4`
`r`	`5`

Symbol Table

4. If a subroutine wants to pass an array instead of an integer, then in C-like languages, we just copy the address of the initial element. What would the pros and cons be of copying the entire array onto the stack instead?
5. If routine P calls routine Q, which of the following would you find or not find in the activation record for the call?

   (a) The return address to the code for P.

(b) The return address from P back to its caller.

(c) A link to the activation record for P.

(d) A link to the activation record for Q.

(e) Space for the local variables of P.

(f) Space for the local variables of Q.

(g) Space for the arguments being passed to Q.

(h) Space for the arguments P received from its caller.

(i) Space for the value that Q will return to P.

(j) Space for the value that P will return to its caller.

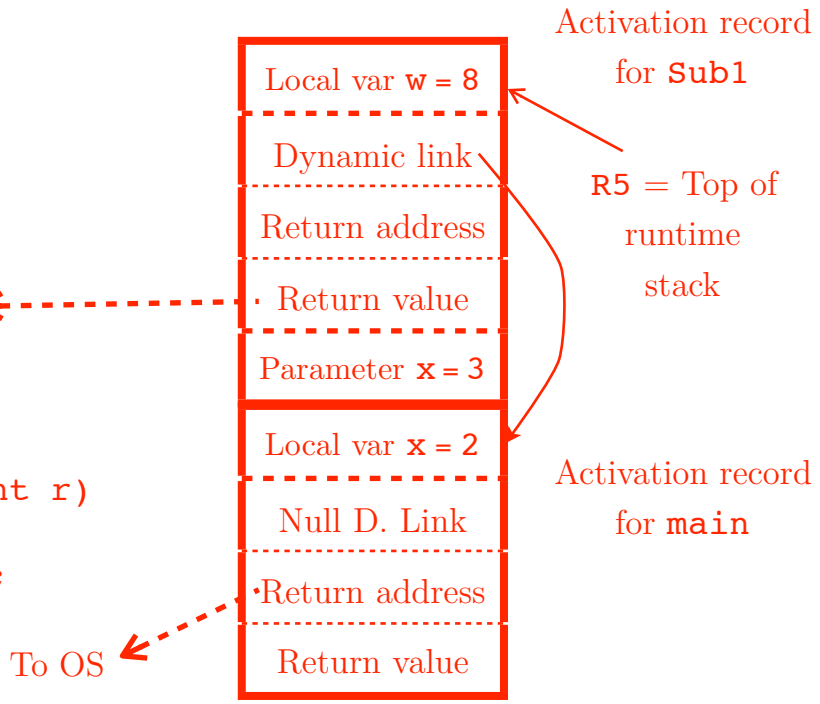
Solution

1. (Note change to value of `w`.)

```
main() {
  int x = 2;
  Sub1(x+1);
}

int Sub1(int x) {
  int w = x+2;
  w = Sub2(w,10);
  return w;
}

int Sub2(int q, int r)
{
  int k; int m[2];
  // *** here ***
  return k;
}
```

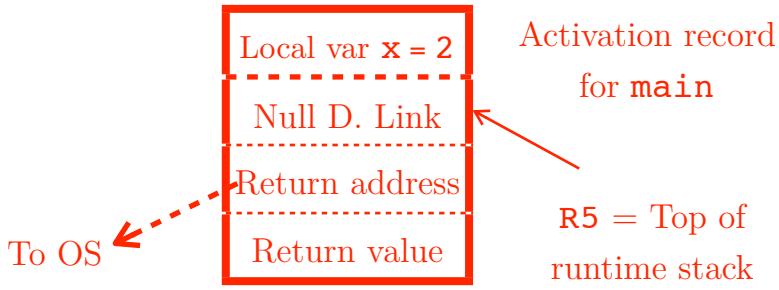


2.

```
main() {
  int x = 2;
  Sub1(x+1);
}

int Sub1(int x) {
  int w = x+2;
  w = Sub2(w,10);
  return w;
}

int Sub2(int q, int r) {
   ...
}
```



3. To implement `k = q+r;` (using `R0` and `R1` as temporary registers)

```
LDR   R0,R5,4    ; R0 = q
LDR   R1,R5,5    ; R1 = r
ADD   R0,R1,R0   ; R0 = q+r
STR   R0,R5,0    ; k = q+r
```

4. If we copy the entire array, then the subroutine can make changes to its parameter without changing the argument array. On the other hand, copying the entire array takes more time and it uses more space on the runtime stack.

5. If P calls Q, the activation record for the call will contain

   (a) The return address to the code for P.

   (c) A link to the activation record for P.

   (f) Space for the local variables of Q.

   (g) Space for the arguments being passed to Q.

   (i) Space for the value that Q will return to P.

