Script generated by TTT

Title: Seidl: Virtual Machines (30.06.2014)

Date: Mon Jun 30 10:25:02 CEST 2014

Duration: 64:50 min

Pages: 34

```
Sema * newSema (int n) {
        Sema * s;
        s = (Sema *) malloc (sizeof (Sema));
        s→me = newMutex ();
        s→cv = newCondVar ();
        s→count = n;
        return (s);
    }
```


The translation of the body amounts to:

alloc 1	newMutex	newCondVar	loadr -2	loadr 1
loadc 3	loadr 1	loadr 1	loadr 1	storer -2
new	store	loadc 1	loadc 2	return
storer 1	pop	add	add	
pop		store	store	
		pop	pop	

The function Down() decrements the counter.

If the counter becomes negative, **wait** is called:

```
void Down (Sema * s) {
        Mutex *me;
        me = s→me;
        lock (me);
        s→count−−;
        if (s→count < 0)   wait (s→cv,me);
        unlock (me);
    }
```

The function Down() decrements the counter.

If the counter becomes negative, **wait** is called:

```
void Down (Sema * s) {
        Mutex *me;
        me = s→me;
        lock (me);
        s→count−−;
        if (s→count < 0)   wait (s→cv,me);
        unlock (me);
    }
```


The translation of the body amounts to:

pop
loadr -1

alloc 1	add	loadc 0		wait
loadr -2	load	less		dup
load	loadc 1	jumpz A		unlock
storer 1	sub	loadr 1		next
lock	loadr -2	loadr -2		lock
	loadc 2	loadc 1	A:	loadr 1
loadr -2	add	add		unlock
loadc 2	store	load		return

The function Down() decrements the counter.

If the counter becomes negative, **wait** is called:

```
void Down (Sema * s) {
        Mutex *me;
        me = s→me;
        lock (me);
        s→count−−;
        if (s→count < 0)   wait (s→cv,me);
        unlock (me);
    }
```


The translation of the body amounts to:

alloc 1	add	loadc 0		wait
loadr -2	load	less		dup
load	loadc 1	jumpz A		unlock
storer 1	sub	loadr 1		next
lock	loadr -2	loadr -2		lock
	loadc 2	loadc 1	A:	loadr 1
loadr -2	add	add		unlock
loadc 2	store	load		return

The function Down() decrements the counter.

If the counter becomes negative, **wait** is called:

```
void Down (Sema * s) {
        Mutex *me;
        me = s→me;
        lock (me);
        s→count– –;
        if (s→count < 0)   wait (s→cv,me);
        unlock (me);
}
```


The translation of the body amounts to:

alloc 1	add	loadc 0		wait
loadr -2	load	less		dup
load	loadc 1	jumpz A		unlock
storer 1	sub	loadr 1		next
lock	loadr -2	loadr -2		lock
	loadc 2	loadc 1	A:	loadr 1
loadr -2	add	add		unlock
loadc 2	store	load		return

The translation of the body amounts to:

alloc 1	add	loadc 0		wait
loadr -2	load	less		dup
load	loadc 1	jumpz A		unlock
storer 1	sub	loadr 1		next
lock	loadr -2	loadr -2		lock
	loadc 2	loadc 1	A:	loadr 1
loadr -2	add	add		unlock
loadc 2	store	load		return

450

The function Up() increments the counter again.

If it is afterwards not yet positive, there still must exist waiting threads. One of these is sent a signal:

```
void Up (Sema * s) {
        Mutex *me;
        me = s→me;
        lock (me);
        s→count++;
        if (s→count ≤ 0)   signal (s→cv);
        unlock (me);
}
```

The translation of the body amounts to:

alloc 1	loadc 2	add		loadc 1
loadr -2	add	store		add
load	load	loadc 0		load
storer 1	loadc 1	leq		signal
lock	add	jumpz A	A:	loadr 1
	loadr -2			unlock
loadr -2	loadc 2	loadr -2		return

The function Up() increments the counter again.

If it is afterwards not yet positive, there still must exist waiting threads. One of these is sent a signal:

```
void Up (Sema * s) {
        Mutex *me;
        me = s→me;
        lock (me);
        s→count++;
        if (s→count ≤ 0)   signal (s→cv);
        unlock (me);
    }
```


The translation of the body amounts to:

alloc 1	loadc 2	add		loadc 1
loadr -2	add	store		add
load	load	loadc 0		load
storer 1	loadc 1	leq		signal
lock	add	jumpz A	A:	loadr 1
	loadr -2			unlock
loadr -2	loadc 2	loadr -2		return

56 Stack Management

Problem:

- All threads live within the same storage.
- Every thread requires its own stack (at least conceptually).

1. Idea:

Allocate for each new thread a fixed amount of storage space.

$\Longrightarrow$

Then we implement:

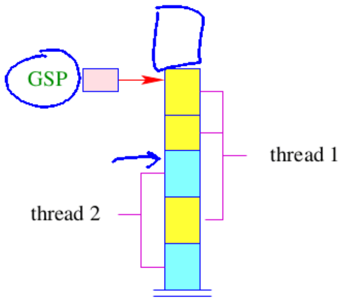
```
void *newStack() { return malloc(M); }
void freeStack(void *adr) { free(adr); }
```

Problem:

- Some threads consume much, some only little stack space.
- The necessary space is statically typically unknown :-(

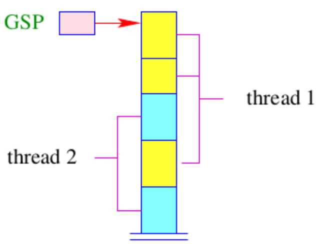
2. Idea:

- Maintain all stacks in one joint Frame-Heap FH :-)
- Take care that the space inside the stack frame is sufficient at least for the current function call.
- A global stack-pointer GSP points to the overall topmost stack cell ...



Allocation and de-allocation of a stack frame makes use of the run-time functions:

```
int newFrame(int size) {
    int result = GSP;
    GSP = GSP+size;
    return result;
    }

void freeFrame(int sp, int size);
```

Allocation and de-allocation of a stack frame makes use of the run-time functions:

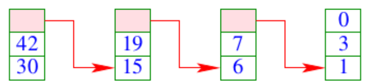
```
int newFrame(int size) {
    int result = GSP;
    GSP = GSP+size;
    return result;
    }

void freeFrame(int sp, int size);
```


Warning:

The de-allocated block may reside inside the stack :-(

$\Longrightarrow$

We maintain a list of freed stack blocks :-)

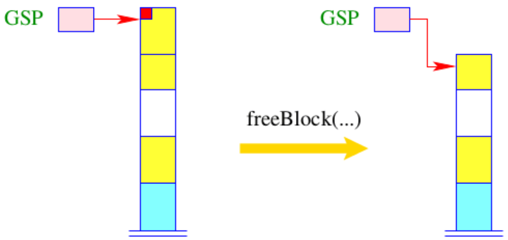


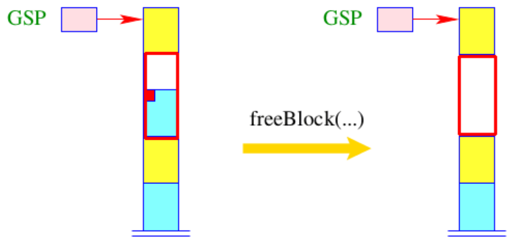
This list supports a function

```
void insertBlock(int max, int min)
```

which allows to free single blocks.

- If the block is on top of the stack, we pop the stack immediately;
- ... together with the blocks below – given that these have already been marked as de-allocated.
- If the block is inside the stack, we merge it with neighbored free blocks:





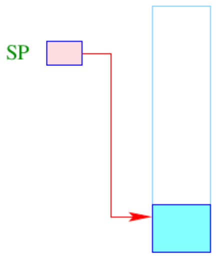
Approach:

We allocate a fresh block for every function call ...

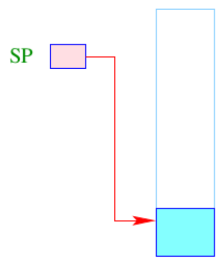
Problem:

When ordering the block before the call, we do not yet know the space consumption of the called function :-(

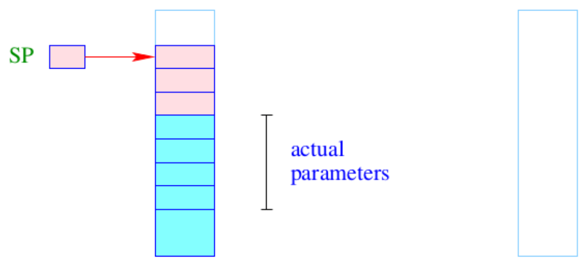
$\Longrightarrow$ We order the new block after entering the function body!



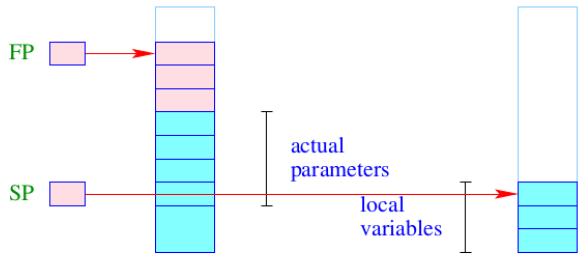
Organisational cells as well as actual parameters must be allocated inside the old block ...



Organisational cells as well as actual parameters must be allocated inside the old block ...



When entering the new function, we now allocate the new block ...



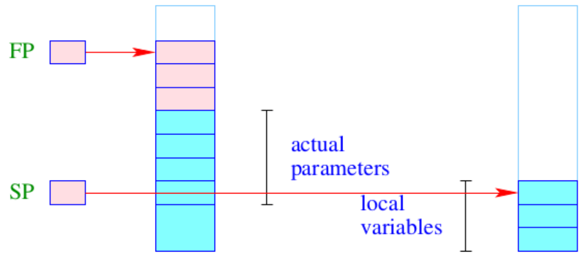
Inparticular, the local variables reside in the new block ...

⟹ We address ...

- the formal parameters relatively to the frame-pointer;
- the local variables relatively to the stack-pointer :-)

⟹ We must re-organize the complete code generation ... :-(

Alternative: Passing of parameters in registers ... :-)



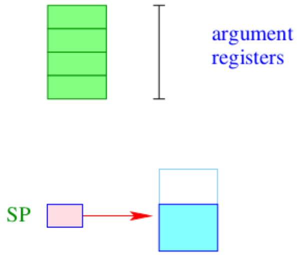
Inparticular, the local variables reside in the new block ...

⟹ We address ...

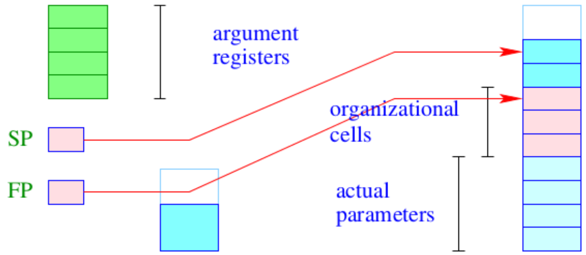
- the formal parameters relatively to the frame-pointer;
- the local variables relatively to the stack-pointer :-)

⟹ We must re-organize the complete code generation ... :-(

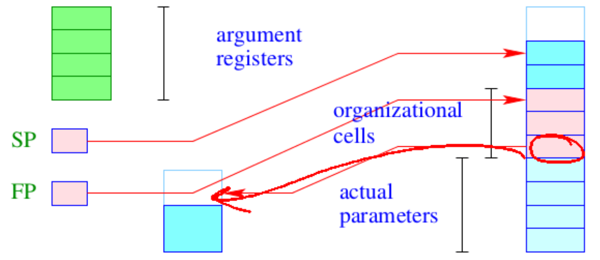
Alternative: Passing of parameters in registers ... :-)



The values of the actual parameters are determined before allocation of the new stack frame.



The complete frame is allocated inside the new block – plus the space for the current parameters.



Inside the new block, though, we must store the old SP (possibly +1) in order to correctly return the result ... :-)

3. Idea: Hybrid Solution

- For the first k threads, we allocate a separate stack area.
- For all further threads, we successively use one of the existing ones !!!

$\Longrightarrow$

- For few threads extremely simple and efficient;
- For many threads amortized storage usage :-))

3. Idea: Hybrid Solution

- For the first k threads, we allocate a separate stack area.
- For all further threads, we successively use one of the existing ones !!!

$\Longrightarrow$

- For few threads extremely simple and efficient;
- For many threads amortized storage usage :-))

468