Fundamentals of Operating System Programming Notes

Lukas Prokop

18th of May 2012

Contents

1 Common pitfalls

via wiki:pitfalls.

Listing 1: Operator precedence pitfall

```
void deallocate(int*** lookupTable)
{
  int i = 0;

  for(; i < MAX; i++) {
    free(*lookupTable[i]);
  }
  free(*lookupTable);
  *lookupTable = NULL;
}
```

The indexing operator has lower precedence than the dereference-operator. Therefore `*(lookupTable[i])` is actually executed, but not intended.

2 git basics

The git graph command (put this into your `\~/.gitconfig` to have `git gr` available):

```
[alias]
  gr = "!git --no-pager log -n 20 --graph --full-history --all --color
       --pretty=tformat:'%x1b[31m%h%x09%x1b[32m%d%x1b
       [0m%x20%s%x20%x1b[33m(%an %ar)%x1b[0m'"
```

2.1 git usecases

- Clone repository from a public URI
- Checkout, list branches and create new ones
- `git show-branch`
- Merges and resolve conflicts
- Fast-forward vs non-fast-forward merge
- `git rebase`

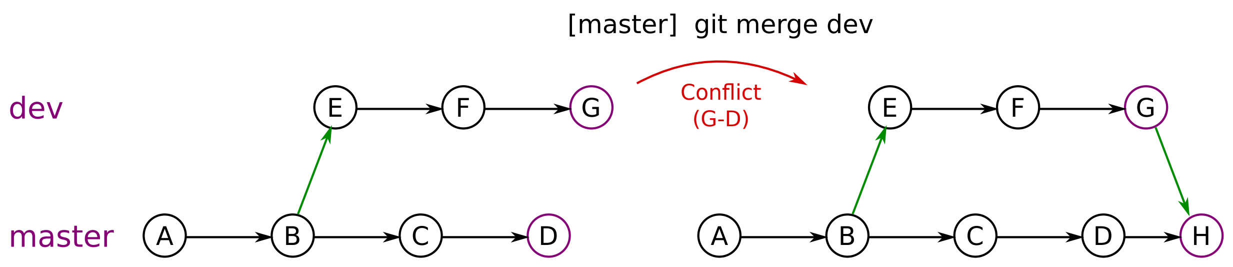

Figure 1: git merge

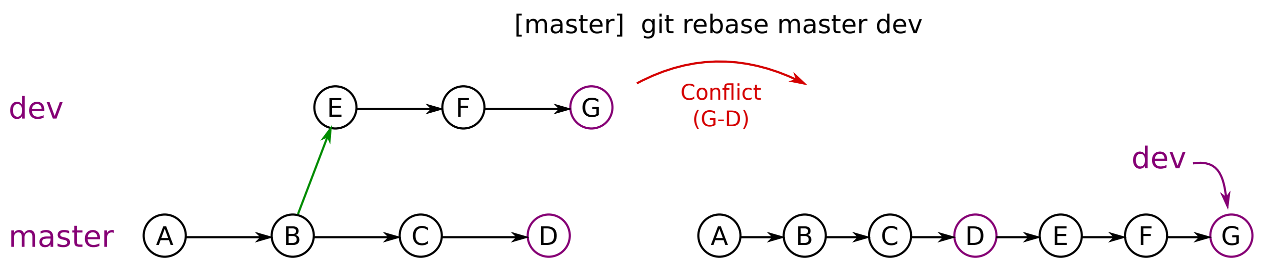

Figure 2: git rebase

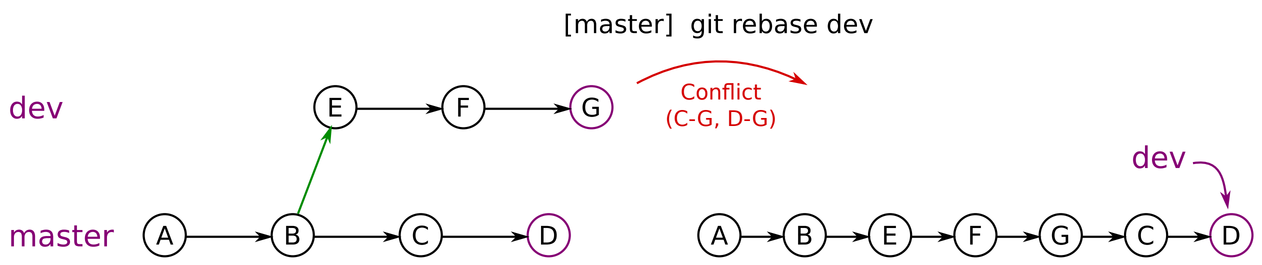

Figure 3: git rebase

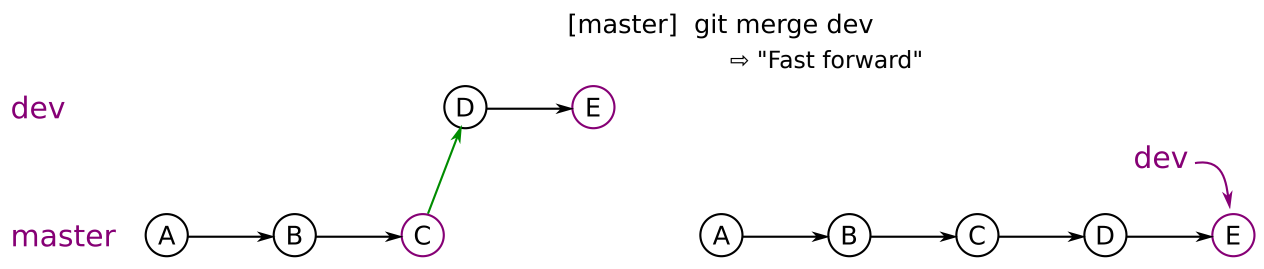

Figure 4: git merge

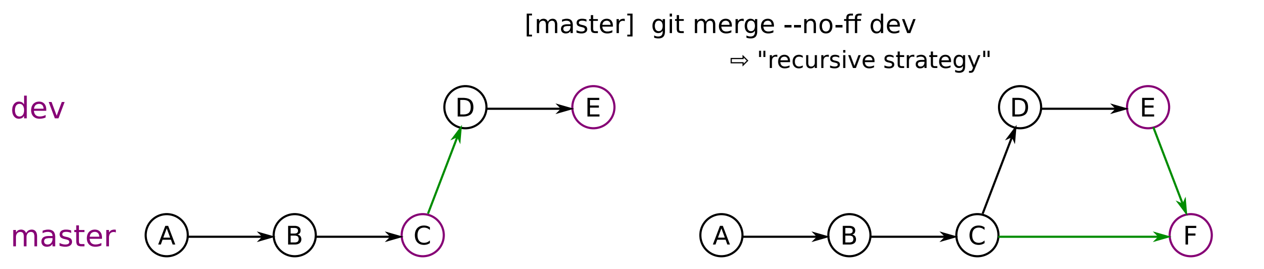

Figure 5: git merge –no-ff

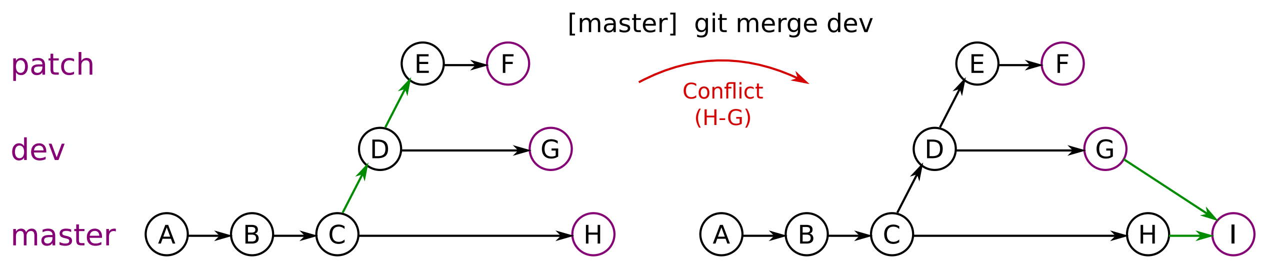

Figure 6: git merge

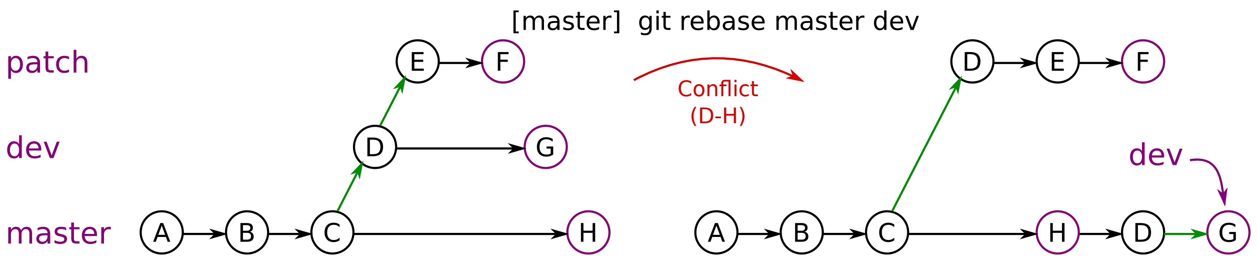

Figure 7: git rebase

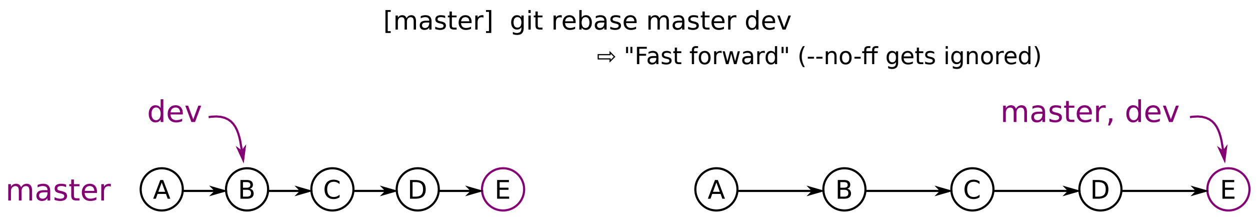
[master] git rebase master dev
⇨ "Fast forward" (--no-ff gets ignored)
dev
master
A
B
C
D
E
master, dev
A
B
C
D
E

Figure 8: git rebase

3 C programming

3.1 C preprocessor token-pasting operator

Listing 2: C program showing usage of token-pasting operator

```
#include <stdio.h>

#define HELLO_WORLD(NUMBER) helloWorld##NUMBER()

void helloWorld1() {
    printf("Hello␣World\n");
}

void helloWorld2() {
    printf("HELLO␣WORLD\n");
}

void main() {
  HELLO_WORLD(1);
  HELLO_WORLD(2);
}
```

3.2 Variadic functions

Listing 3: Example of a variadic function

```
bool solve(int argc, bool arg1, ...)
{
  // example usage:
  //   solve(1, true, And, solve(1, false, Or, false))
  va_list ap;
  va_start(ap, arg1);
  int i;
  logic_function operation = NULL;
  bool arg2;

  for (i=0; i<argc; i++)
  {
    operation = va_arg(ap, logic_function);
    arg2 = va_arg(ap, bool);
    arg1 = operation(arg1, arg2);
  }
  va_end(ap);

  return arg1;
}
```

3.3 Function pointers

Listing 4: Function pointers example

```
#include <stdio.h>

int function(int argc, char **argv)
{
  unsigned int x;
  for (x=0; x<argc; x++)
  {
    printf("%s.\n", argv[x]);
  }
  return argc;
}

void call(void *function)
{
  char *argv[1] = {"Hello␣World"};
  int (*ref) (int, char**) = function;
  printf(">␣%d\n", ref(1, argv));
}

int main()
{
  int argc = 2;
  char *argv[2] = {"Hello", "World"};

  // store function pointer in a variable
  int (*f_ref) (int, char**) = &function;

  // call function pointer
  f_ref(argc, argv);
  // function pointer as parameter
  call(&function);

  return 0;
}
```

3.4 C++ templates

The following methods are required for a class to be used as a container element (”value”):

- copy constructor
- assignment operator
- (probably: default constructor)

These requirements have to be fulfilled to be used as a key in a container:

- copy constructor
- assignment operator
- less-than operator
- (probably: default constructor)

3.5 pthreads

If two threads want to communicate via global variables and one of the threads cannot reset the value back to its neutral value, we can take the following approach:

The value v gets two counters assigned for each thread (c_1, c_2). If value v gets set to a (and $c_1 = 0$), the second thread recognizes this change because of the different value a in v. Because we actually don’t know how many times the value v was set to a, we need to read c_1. Each time the value was set, c_1 got incremented. So we will set $c_2 = 0$ and increment this value until $c_1 = c_2$. This way we can recognize events occuring only in 1 thread.

Listing 5: Example of POSIX threads

```
  // create a thread and whatever you need..
  pthread_t thread;
```

```
pthread_create(&thread, NULL, input_thread, &comm);

// ..

// clean up.
pthread_join(thread, NULL);
```

4 POSIX

5 UNIX shells

`A &` Execute `A` in a separate process

`A > B` Redirect stdout of `A` to file `B`

`A >> B` Append stdout of `A` to file `B`

`A | B` Redirect stdout of `A` to `B` (piping)

0 is stdin, 1 is stdout, 2 is stderr. So `A 2>&1` means stderr will redirected to the place where stdout is pointing to.

6 Ressource sharing

- Mutex
- Semaphore
- Condition variables
- Shared Memory via shm_*

7 Memory management

- The current memory usage of a process can be dumped by catting file `/proc/self/status`
- realloc, malloc and free can be reimplemented by the brk/sbrk syscalls

- void pointer arithmetic works bytewise (void pointer plus one points to the next byte)
- A structure in C always gets aligned to full bytes (12 bits become 16 bits = 2 bytes)
- A structure in C gets left aligned. Therefore the alignment bits are inserted at the very beginning of the structure.
- printf with %p prints pointers bytewise. One number higher means one byte more.

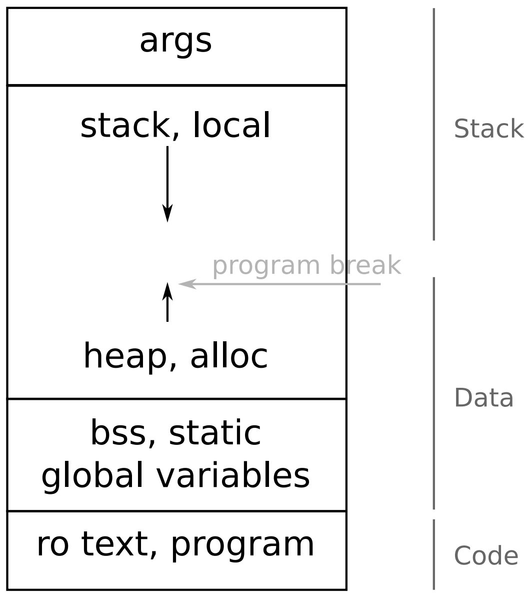

Figure 9: Memory management

8 x86 assembly in GAS syntax

(a) Dereference memory address a

movl $5, %eax Register EAX will be set to 5

9 GDB usage

x 0xADDRESS examine (print value at address)

bracktrace print last calls and their parameters or current state in execution

run Run the program

step Step by step execution of instructions (in difference “next” will step over instead of into functions)

b X Print register

p $eax Print register EAX