Karlsruhe Institute of Technology

Operating Systems 2016/17
Tutorial-Assignment 2

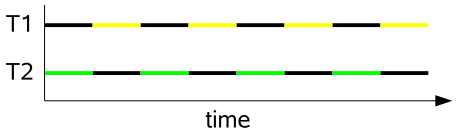
Prof. Dr. Frank Bellosa
Dipl.-Inform. Marc Rittinghaus

Question 2.1: Multi-Programming

a. Explain the difference between single-programming and multi-programming systems. What is the advantage of multi-programming?

Solution:
Single-programming, *also known as uni-programming, allows only one process to run at a time.* Multi-programming *is a system feature enabling* $n > 1$ *processes to execute concurrently, sharing the physical resources. The standard solution to make this possible is preemption. Every process may run for a while. A timer interrupt (or any other interrupt in tickless kernels) will then trap into the kernel which will run a routine to put the currently running process to sleep, saving its processing state. Then the processor is given to another process which may run in turn.*

A benefit of multi-programming can be that the execution of different processes may be interleaved on the CPU. Suppose the CPU phases (black lines) are as long as the I/O phases, and suppose that we have to execute two different processes. The figure shows a possible schedule if T1 only uses I/O device x *(yellow line) whereas T2 always uses I/O device* y *(green line),* $x \neq y$*:*



In a single-programming system there would be no overlapping on the CPU. So, the CPU would be idle up to 50 % of the time while accessing the I/O device.

Note that interrupts are required so that a device can notify the CPU once a request is completed. Without interrupts, the CPU would have to poll the device, and thus would not be able to do other work in the meantime.

Many issues arise when using multi-programming systems. Most importantly protection: How do you separate activities cleanly (processes serve this purpose)? But also issues such as fairness and accounting.

b. Explain the difference between CPU-bound and I/O-bound processes.

Solution:
A CPU-bound process is a process that rarely invokes I/O-operations. As a consequence, CPU-bound processes are not very likely to block (i.e., they have to wait for an I/O operation to complete and thus cannot continue executing on the CPU). In contrast, I/O-bound processes perform only short computations between I/O-jobs, thus, they are expected to block relatively often.

c. Why is a good mixture of CPU-bound and I/O-bound processes preferable?

Solution:
A good mixture of I/O-bound and CPU-bound processes ensures that both the I/O-devices and the CPU are utilized.

If the system executes an I/O-bound process, the probability that the process will soon block and wait for I/O completion is high. This phase is ideal to run a CPU-bound process, because now the I/O device is busy but the CPU idles. If we had only I/O-bound processes, we would have much idle time for the CPU, because all processes would be waiting for I/O, potentially even slowing each other down by stressing the I/O device. On the other side, having only CPU bound processes leaves the I/O devices idle and puts all burden on the CPU.

Question 2.2: Processes in Unix

a. What keeps a process from accessing the memory contents of another process?

Solution:

Every process lives in its own address space, which means that every process has it's own view on the memory it uses. Accessing an address (e.g., `*(char*)0x1234`) will (most likely) lead to different results in different processes.

Address spaces are protection domains: Program code residing in one address space cannot access data from another address space (unless the kernel supports sharing and both sides agree to share some data).

The protection is implicit: If you can't name it you can't touch it.

The address translation is done by the hardware (e.g., MMU) and the OS. It will be covered in great detail later in this lecture.

b. What are typical regions in a process address space? What is their purpose?

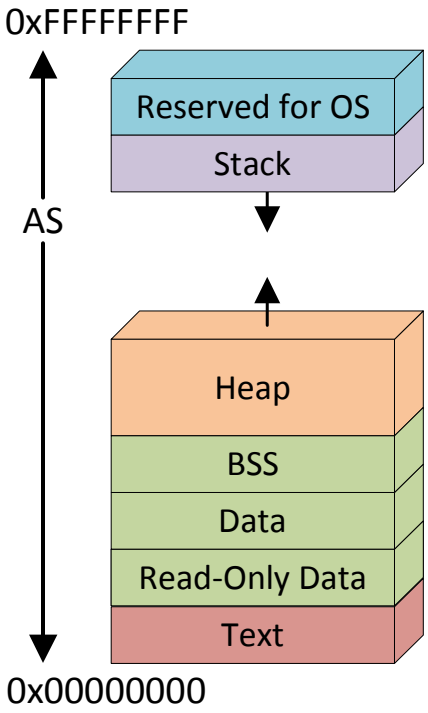
Solution:
From high addresses to low addresses:

OS *The address range of the kernel is usually at the top end of the address space covering the high addresses; e.g., the top 2 GiB in a 4 GiB (i.e., 32 bit) address space. This region is shared across all address spaces and contains kernel code and data. It may not be accessed by user mode code.*

Stack *The stack segment provides the temporary memory for program execution necessary to hold local variables, function call parameters and return addresses. It is one of the most important address space ranges. The stack is usually located at high addresses and grows downwards as the function call depth increases. Depending on the platform, the program binary may specify a start size for the stack.*

Heap *The heap provides space for dynamically allocated data. This area is usually managed by a heap allocator, which is implemented as a user space library. The heap allocator first retrieves a huge memory chunk from the operating system and then divides this chunk into smaller pieces as required by subsequent calls to `malloc()/free()`. A call to `malloc()` is therefore usually very fast because it does not require contacting the OS kernel. The heap is process private.*

BSS *The BSS segment (**b**lock **s**tarted by **s**ymbol) is reserved for data that is uninitialized at program start. The operating system usually initializes this range to zero. The program binary only informs the loader about the starting address and size of the area, but*



it does not explicitly contain the 'zero'-data and thus does not take up space in the program binary. This area is private to each process because it may be modified during runtime.

Data *The data segment holds pre-initialized data that can be modified during program execution. Global variables that have a default value fall into this range. This area is loaded from the program binary file but then remains private to the current process.*

RO Data *As the data segment, the* ***r****ead-****o****nly data segment contains pre-initialized data. However, this data may not be modified during execution. An example are strings that are passed to* `printf()`*. This area is loaded from the program binary and due to its read-only nature can be shared across all processes that execute the same program.*

Code/Text *The text segment contains the program code of a process's executable. The instruction pointer of the CPU points to the current instruction in this section, when the program executes in user mode. This area is loaded from the program binary file and is usually shared across all processes that execute the same program.*

c. What does the `fork()` system call do?

Solution:

`fork` *creates a child process that is identical with the original process (i.e., the one that invoked* `fork()`*) in most parts: Although both parent and child possess an own address space, they have the same address space layout and data (as if the parent address space would have been copied). They also share open files. There are, however, some exceptions: The newly created process has its own, unique process id, and its parent id is set to the id of the parent process.*

For a more detailed overview of differences, you are encouraged to have a look at the respective man page (i.e., `man fork`*).*

d. Write a small C program that creates a child process. Each process shall print out who it is (i.e., parent or child). The parent shall also print out the child's PID and then wait for the termination of its child.

Solution:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

int main()
{
    pid_t pid;
    switch( (pid = fork()) )
    {
    case -1:
        printf( "Error. Fork failed\n" );
        break;
    case 0:
        printf( "I am the child!\n" );
        break;
    default: // pid > 0
        printf( "I am the parent!\n" \
                "Child PID is %d\n", pid );
        wait( NULL );
        printf( "Child terminated\n" );
    }

    return 0;
}
```

e. Assume you have to write a shell that can be used to launch arbitrary other programs. Is the `fork()` system call sufficient for that purpose?

Solution:
`fork()` is insufficient, as it only creates a copy of the originating process. `execve()` can be used to replace the currently running program with another, for example to load a new binary into the **current** *address space. When a shell creates a new process to execute some program, it will first fork itself and then invoke `execve()` within the child process to replace the shell code with the code of the program that shall be executed.*

Question 2.3: Stacks and Procedures in C

a. Discuss the following code fragment. Try to visualize the stack contents before, during, and after the execution of `foo`. All values are passed via the stack between calling and called function. An `int` is 4 bytes and a `double` is 8 bytes long. Assume a 4-byte aligned, downwards growing pre-decrement stack and the existence of a stack-frame pointer. All local entities within a function are addressed relative to this frame pointer.

```
double foo ( int *p )
{
    int x;
    double y;
    x = *p;
    // do something useful
    return y;
}
```

```
double bar ()
{
   double d;
   int i = 42;
   d = foo( &i );
   return d;
}
```

Solution:
Different solutions are possible; we assume a call sequence that first pushes all arguments onto the stack, then allocates some space for the return value, and finally calls the function. In pseudo code (argument sizes assumed to be 4):

```
push  argN          # push arguments in reverse order
...
push  arg1          # first argument is pushed last
add   SP, -r        # r: size of func's return value, may be 0
call  func          # pushes return address (RA) on the stack
load  reg, (SP)     # retrieve return value into reg
add   SP, r+4N      # remove call frame: return value and arguments
```

The called function sets up its frame, accesses the arguments and stores its return value as follows:

```
push  FP            # save old frame pointer
move  FP, SP        # FP := SP, set up own frame pointer
add   SP, -n        # n: size of all local variables
load  reg, (FP)+8+r # first argument (skip old FP, RA, and return value)
...
store reg, (FP)+8   # store return value (skip old FP and return address)
move  SP, FP        # SP := FP, remove local variables
pop   FP            # restore old frame pointer
return              # pops return address from stack
```

Figure 1 shows the stack at four distinct points in time. After having returned from the called function, the caller must retrieve the return value (if any) and then clean up the stack (remove the previously pushed arguments from the stack; in this example by subtracting 12 from SP).

Note 1: After the call, the frame of the caller is still accessible, but **should** *not be accessed directly!*

Note 2: After returning, the frame of the callee is not cleared automatically, but only marked as "free" (beyond current stack pointer)!

Conclusion: Using a stack for parameter passing has two main advantages:

(a) The context of each callee (its frame) is automatically pushed onto and popped from the stack, no additional actions are needed.

(b) The parameters being pushed onto the stack are accessed via the stack frame pointer, thus you can implement procedures with a variable number of parameters very easily.

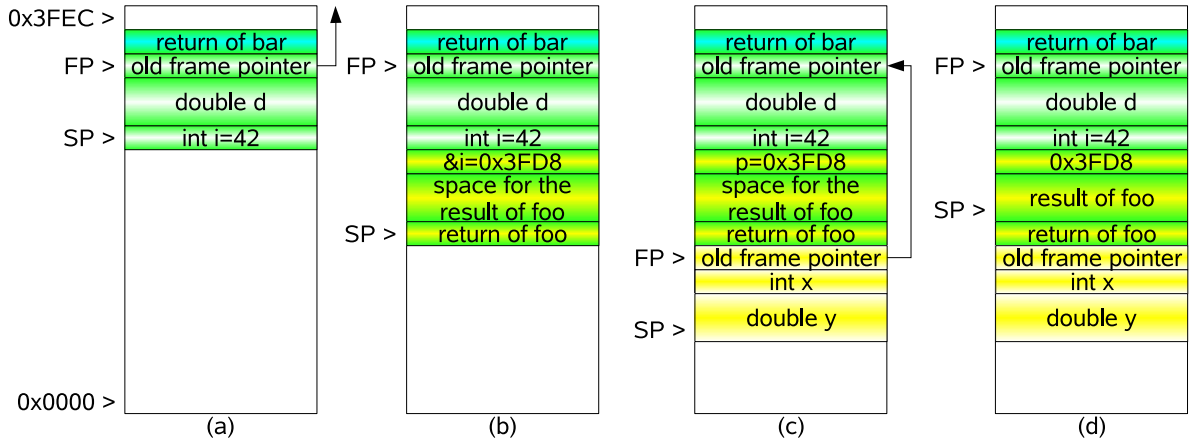


Abbildung 1: Stack layout (a) before starting to call `foo`, (b) right after the `call` instruction has been executed, (c) during the execution of `foo`, and (d) right after the `return` instruction in `foo` has been executed. FP=frame pointer, SP=stack pointer

b. Characterize as briefly and precisely as possible the difference between a function and a macro in C. Outline the consequences of these two different implementations with regard to the caller.

Solution:
The code of a function exists only once, independent of how often the function is being called (exception: inlined functions). For macros, each call results in its respective code being inserted at the call-site (i.e., macros are always inlined). In C, macros are expanded in the source code by the pre-processor.

As a consequence, code with many macros usually results in larger code than the same code using functions. Another effect is to be seen in the runtime behavior: function calls incur a small overhead compared to macros for jumping to and back from the called function.

An important point to note is that macros are more susceptible to side-effects than functions. While a function `int max(int x, int y) { return ((x > y) ? x : y); }` *always works as expected, the similar macro* `#define MAX(x,y) (((x) > (y)) ? (x) : (y))` *does not:* `int a=4, b=3; MAX(a++, b)` *surprisingly yields* `5` *and increases* `a` *to 6, whereas* `int a=4, b=3; max(a++, b)` *produces the expected value of* `4` *and increases* `a` *only once so that* `a == 5` *holds afterwards. (Figure out how the macro is evaluated!)*

Last but not least, macros are untyped: The compiler does not check whether the macro arguments have a specific type. Where the above function forces the use of a `int`*, the macro can also deal with floats. This property is both a feature (great flexibility) and a well-known source of errors.*