Operating Systems – sprint 2022
Tutorial-Assignment 2

Instructor: Hans P. Reiser

Question 2.1: Multi-Programming

a. Assume that process P_1 is currently running on the (single) CPU of a system. Explain all events that may lead to the execution of P_1 being replaced by the execution of a different process P_2.

Solution:
First, some event that causes a kernel entry (i.e., context switch from user mode to privileged mode) needs to happen. This can be any of the events discussed in the previous tutorial:

- *A system call invoked by P_1*
- *An exception caused by the execution of P_1*
- *An external interrupt*
- *Note: You could list additional events, such as the termination of P_1 or P_1 explicitely yielding the execution, but all these operations are again system calls (exit(), yield()...)*

After any kernel entry, the scheduler may then decide to activate a different process instead of returning to the previously running process.

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 (blocking means 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 it usually a good strategy for a CPU scheduler to favour I/O-bound processes over CPU-bound processes?

Solution:
I/O-bound processes are characterized by short bursts of CPU utilization interleaved with longer periods of waiting on I/O operations.

- *Favouring I/O-bound processes will assure good utilization of I/O devices (the sooner the I/O-bound processes gets the chance to run on the CPU, the sooner it will issue the next I/O operation, avoiding that the I/O device remines idle for longer periods of time*
- *I/O-bound processes will use the CPU only for short burts, so favouring them will not actually give them much more CPU time than a CPU-bound process. The impact on CPU-bound processes is likely to be only small.*
- *Interactive processes are typically I/O-bound and the system will be more responsive (from an interactive user point of view).*
- *Faviouring I/O-bound processes allows the system to react faster upon I/O events, such as reading I/O buffers, und thus decreases the probabiliy of occurences such as packet loss due to lack of space in I/O buffers.*

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. (to "name" here means to have an address in your virtual address space that translates to the physical memory address your want to access)

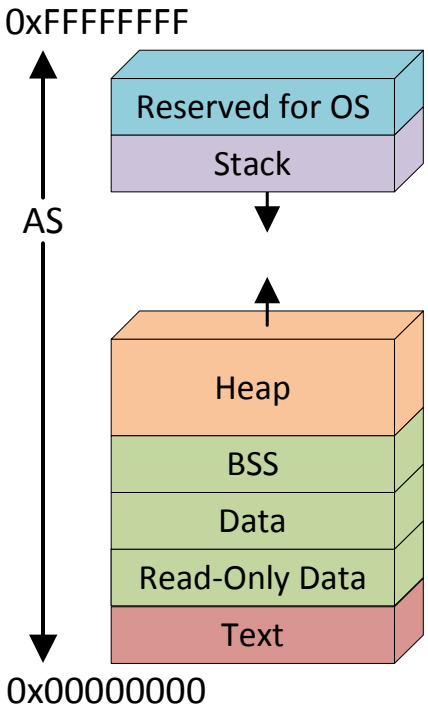
The address translation is done by the hardware (e.g., MMU) and the OS. It will be covered in great detail later in this lecture.

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. (Since Meltdown (`https://meltdownattack.com/`) chances are good that your kernel uses Kernel Page Table Isolation (KPTI), in which case the kernel code and data is removed from the address space when running in user mode)*

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. *Preliminary notes: You should remeber from the introduction to C programming that local variables of a function are placed on the stack. Unlike static global variables, which during the execution of an application are always at the same location in memory, the address in main memory (on the stack) of such local variables of a function might be different for each invocation of that function (depening on what data currently exists on the stack when the function is invoked).*

Nevertheless, for accessing such a variable, the CPU needs to know its address. A very common approach for implementing local variables is the use of a stack-frame pointer. With this approach, when entering a function:

- *The current value of the frame pointer (FP) of the previous function is saved (for example on the stack).*
- *The current value of the stack pointer (SP) is copied to the frame pointer register (FP)*
- *The stack pointer is decremented by the size of all local variables.*

Any time within the execution of the function, the local variables can found relative to the FP. For example, if there are two local variables of type uint32_t, they can be found at address (FP)-4 and (FP)-8. Likewise, if the caller passes arguments on the stack, these can be found relative to the BP as well. For example, assuming a 32-bit architecture, you could find the saved previous frame pointer at address (FP), the return address of the caller at (FP)+4 and a function argument at (FP)+8.

Discuss the following code fragment. Try to visualize the stack contents before `foo` calls `bar`, as well as 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.

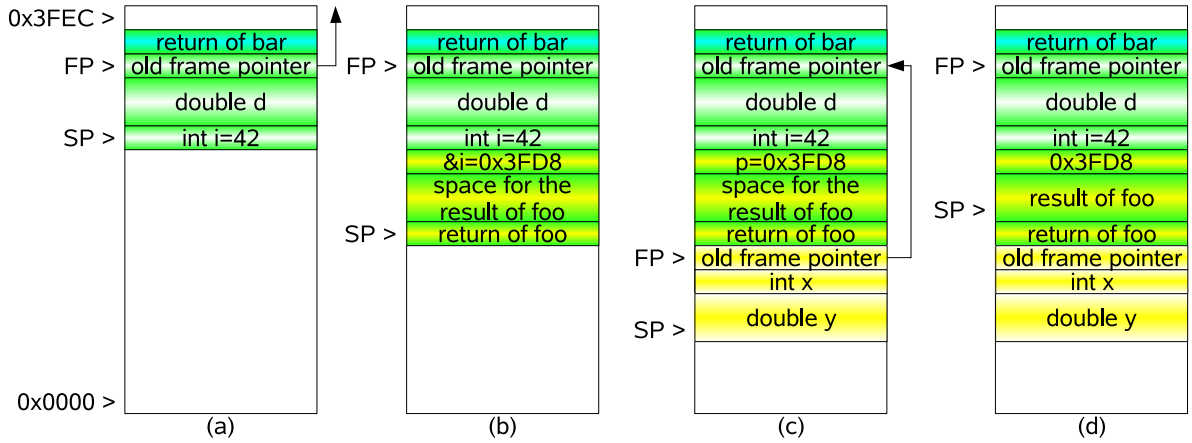


Abbildung 1: Stack layout (a) before starting to call `foo`, (b) right after the `call` instruction has been executed, (c) during the execution of `foo`, and (d) right after the `return` instruction in `foo` has been executed. FP=frame pointer, SP=stack pointer

Question 2.4: Dynamic memory management in C

a. malloc() and free()

`malloc()` is a C library function to dynamically allocate memory on the heap. The function works fully in user space, using memory that was previously allocated by the operating system for the heap of an application.

`malloc()` allocates memory of a specified size (in bytes) and returns a pointer to the beginning of the allocated block. `malloc()` does not know the type of data we are going to store in that memory, and thus the type of the pointer it returns is `void *`, a pointer with no type information. In order to access the pointer, we need to *cast* it to the type we want to use.

```
#include <stdlib.h>
#include <stdio.h>

int main() {
        int *intPtr;

        intPtr = (int *)malloc(sizeof(int));
        if(intPtr == NULL) { printf("malloc failed\n"); exit(1); }

        *intPtr = 42;

        printf("The value is %d\n", *intPtr);
        free(intPtr);
}
```

Extend the following program (calculating prime numbers) such that the required memory for the `primes` array is dynamically allocated, corresponding to the maximum number provided as command line argument.

```
#include <stdio.h>
#include <stdlib.h>

// First argument argv[0] is the program name, argv[1],... the real arguments
// argc is the total number of arguments, including the program name (i.e., it is always at least 1)
```

```
int main(int argc, char *argv[])
{
    int i, j, max;
    int *primes = NULL; // size not known at compile time, needs to be allocated dynamically

    if(argc<2) { printf("Usage: %s <number>\n", argv[0]); exit(1); }

    max = atoi(argv[1]);
    printf("Finding prime numbers from 1 to %d\n", max);

    ////
    ////////////////////////////
    //// ADD YOUR CODE HERE
    ////////////////////////////
    ////

    //populating array with naturals numbers
    for(i = 2; i<=max; i++) primes[i] = i;

    //standard prime number sieve
    for(i=2; i*i <= max; i++) {
        if (primes[i] != 0) {
            for(j=2; j<max; j++) {
                if (primes[i]*j > max)
                    break;
                else
                    primes[primes[i]*j]=0;
            }
        }
    }

    for(i = 2; i<=max; i++) {
        if (primes[i]!=0)
            printf("%d\n",primes[i]);
    }

    // All memory of a process will be freed in any case if the process terminates.
    // But in all other cases, make sure to free memory previously allocated with malloc,
    // as soon as you don't need that memory anymore.
    free(primes);
    return 0;
}
```

Solution:

```
primes = (int *)malloc( (max+1) * sizeof(int) );
if (!primes) {
  printf("Out of memory error");
  exit (1);
}
```

Question 2.5: Call by reference in C

a. Consider the following short C program. Does it print 12, 42, or another value? Explain why!

Solution:
It prints 12. Function arguments in C are always passed by value. The function `update_value` *receives the value 12 (on the stack or in a CPU register, depending on the calling conventions used), and assigns it to a local variable of the function. Changes to that variable (which is placed on the stack) do not affect other variables outside of that function.*

```
#include <stdio.h>

void update_value(int val) {
    val = 42;
}

int main() {
    int value = 12;
    update_value(value);
    printf("value is %d\n", value);
}
```

b. What needs to be changed such that the `update_value` function updates the variable `value` in the main function?

Solution:
We can have call-by-reference calling semantics in C by passing a pointer to variable. The value we pass to the function now is NOT the value (12), but instead the address of the memory location that containes the value 12. By dereferencing that pointer, `update_value` *can modify the variable in the* `main` *function.*

```
#include <stdio.h>

void update_value(int *val) {
    *val = 42;
}

int main() {
    int value = 12;
    update_value(&value);
    printf("value is %d\n", value);
}
```

c. Now consider this example where the basic type of the variable we want to update is not an integer, but a pointer. Explain what needs to be changed such that the program actually prints the value selected by `update_value()`?

```
#include <stdio.h>

void update_value(char *val) {
    val = "YES";
}

int main() {
    char *answer = "NO";
```

```
    update_value(answer);
    printf("My answer is %s\n", answer);
}
```

Solution:
*Similar to part (b), we again need to pass a parameter "by reference", but in this case the paramter we pass is itself a pointer. In this case, we thus get a pointer to a pointer (to a sequence of caracters). In essence, the solution is the same as in (b), The parameter passing ("&" operator) and the dereferencing for access is exactly the same, but note that the type of the paramter is now "char **", a pointer to a pointer to a char.*

```
#include <stdio.h>

void update_value(char **val) {
    *val = "YES";
}

int main() {
    char *answer = "NO";
    update_value(&answer);
    printf("My answer is %s\n", answer);
}
```