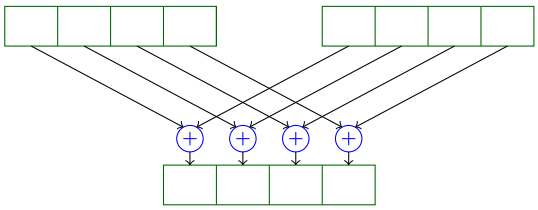
CS4375 Operating Systems Concepts
Fall 2024. Midterm II
11/13/2024

You need to turn in the theoretical part of the exam (Sections 1 and 2) by 11/13/2024 11:50AM MST on paper, directly to your instructor. You need to turn in the practical part of the exam (Section 3) by 11/13/2024 11:59PM MST by email to `utep-os-fall-2024-midtermII@christoph-lauter.org`*. You need to turn in the C source code file* `server.c`*. You can use the dandelion server for development. Programs that do not compile will receive very little credit. Do not forget to put your name on all sheets of paper you hand in and into the first couple of comment lines in the source code file* `server.c`*. Code is provided for the client code that goes with the server but no boilerplate code is provided for the server.*

1 Scatter-gather SIMD Instructions and Page Faults (10pts)

Computers are used to process data streams that often times come in groups: stereo music has a left and a right channel, or 3D movie decoding needs to process both the left-eye and the right-eye datastream. Other problems are called *embarrassingly parallel*, as they naturally come as several streams of data-processing tasks with no interdependence. The Basic Local Aligment Search Tool (BLAST) in bioinformatics is an example algorithm that is embarrasingly parallel.

In order to provide appropriate hardware for such parallel computing tasks, CPU vendors have invented Single Instruction Multiple Data (SIMD) instruction set extensions: single instructions work on a vector (i.e. short array) of data points in a parallel fashion:



Examples of such SIMD instruction sets include Intel MMX/SSE/AVX or ARM neon. With the latest AVX-512 instructions, SIMD instructions work on 512bit wide vectors.

In a SIMD instruction set, all registers contain vectors representing short arrays of elements. The register width is commonly fixed; the number of elements in a register varies with the size of an element. For example, an AVX-512 register can hence be split into eight 64bit variables, 16 variables of 32bit width or even 64 byte-wide variables.

As computing purely in registers does not make sense, SIMD instruction sets all provide instructions to load SIMD registers from memory and to store their content back to memory. Classical SIMD load/store instructions work with only one address, i.e. the address to load from or to store to comes from a regular register, not a SIMD register. A C emulation of an AVX-512 store for 64bit values could be

```
void avx_512_64bit_store(void *ptr, uint64_t vect[8]) {
  int i;
  for (i=0;i<8;i++) {
    ((uint64_t *) ptr)[i] = vect[i];
  }
}
```

In this C emultation code, the `uint64_t vect[8]` stands for the SIMD register, which is not stored in main memory in the actual hardware implementation.

Assuming that `ptr` is an arbitrary pointer inside the running process' address space and `vect` is a register, explain how many page faults can maximally occur with 4096 byte wide pages, *after instruction fetch and decoding have taken place*.

Assume now that `ptr` is an integer multiple of 64 and remember that $512/8 = 64$. Such an additional requirement is called a *vector alignment property*. Explain how this reduces the maximum number of possible page faults. You may use a drawing to illustrate your point.

Unfortunately, certain applications need more than load and store instructions, for which only one address can be specified. For this reason, the most recent SIMD instruction sets also include so-called *scatter-gather* instructions. A *scatter instruction* works on two SIMD registers. The one register contains pointers, the other register contains the data. The instruction scatters the data across the whole memory space, storing the i-th element of the data register at the corresponding i-th address coming from the address register. A C emulation of an AVX-512 scatter could be

```
void avx_512_64bit_scatter(void *ptr[8], uint64_t vect[8]) {
  int i;
  for (i=0;i<8;i++) {
    *((uint64_t *) (ptr[i])) = vect[i];
  }
}
```

A *gather instruction* does the opposite of a scatter instruction, loading the different vector elements of a SIMD register from the various addresses specified by an address SIMD register. It hence gathers data.

Assuming that `vect` is a SIMD register and that each of the addresses `ptr[i]` is an integer multiple of 8, i.e. that the accesses are aligned, explain how many page faults can maximally occur on the above defined scatter instruction with 4096 byte wide pages, *after instruction fetch and decoding have taken place*. Also compute the maximum number of possible page faults if the addresses `ptr[i]` are unaligned, i.e. if they are arbitrary addresses.

As always, the use of drawings, along with textual explanations, is recommended for this type of questions.

2 Virtual and Physical Addresses (40 pts)

Assume a 32bit little-endian system with 32bit virtual and physical addresses, using 4096byte wide ($2^{12} = 4096$) pages and two levels of 1024-entry page tables ($2^{10} = 1024$). Each entry of the tables consists of a 32bit physical address to the next table resp. to the physical page and of a 32bit entry with flags (in this order). The least significant bit in that flags part indicates whether the page is mapped in (if the bit is set to one) or mapped out (if the bit is set to zero). Each entry in the tables is hence 8bytes wide; remember that $8 = 2^3$.

Below you see a table with an extract of the system's physical memory[1]. Let the physical base address of the first page table be `0xcafe0000`.

Use the memory extract and the base address to translate the virtual address `0x52fa8eef` to a physical address. If you cannot perform this translation because the page is not mapped in, indicate that a page fault

[1]Everything that starts in `0x` in this exercise is notated in hexadecimal (base 16). Like always, in this notation, the most significant digit is to the left, the least significant to the right.

occurs. In your answer, detail each step of the translation; do not just give the final translation result.

Remember that the system is little-endian; this means given an address, e.g. `0x12345600`, you can find the least significant byte `0x88` of the (example) 64bit value `0xaabbccddeeff9988` at the address `0x12345600`, the next byte `0x99` at `0x12345601` and the most significant byte `0xaa` at `0x12345607`.

Address (32bit)	*Content (64bit)*							
	@+0	*@+1*	*@+2*	*@+3*	*@+4*	*@+5*	*@+6*	*@+7*
0xbeef9390	0x1f	0x2a	0x34	0x42	0xea	0xde	0xca	0xfe
0xbeef9398	0x22	0x00	0x11	0x3f	0x3a	0x42	0x19	0xca
0xbeef93a0	0x00	0x00	0xbb	0xde	0x24	0x34	0x10	0xa3
0xbeef93a8	0x00	0xb0	0xbe	0xba	0x07	0xa0	0xff	0x42
0xbeef93b0	0x2a	0x20	0xbe	0xbe	0x02	0x30	0x23	0x17
0xbeef93b8	0x00	0x1a	0x1a	0x00	0xa0	0x20	0x00	0xff
0xbeef93c0	0x00	0xb0	0xbe	0xba	0x06	0xa0	0xff	0xa4
⋮	⋮							
0xbeefad30	0x00	0x80	0x12	0x33	0x12	0x20	0xfe	0xc4
0xbeefad38	0x00	0xc0	0xaa	0xf7	0x16	0x42	0x00	0x2b
0xbeefad40	0x00	0xb0	0xad	0xde	0x17	0x42	0xff	0x2a
0xbeefad48	0x00	0xb0	0xad	0xdd	0x18	0x43	0x0f	0x07
0xbeefad50	0xb8	0x80	0xde	0xad	0xca	0xfe	0xaf	0xfe
0xbeefad58	0xb1	0x6b	0x00	0xb5	0x42	0x00	0x00	0xf0
0xbeefad60	0x00	0xfe	0xfa	0xad	0xde	0x02	0x00	0x01
⋮	⋮							
0xcafe0140	0x20	0x00	0xea	0xad	0x02	0xfe	0x07	0x73
0xcafe0148	0x80	0xff	0x00	0x00	0x00	0xfe	0xff	0x17
0xcafe0150	0x42	0x23	0xc2	0xad	0xde	0x24	0x19	0x88
0xcafe0158	0x23	0x00	0x42	0x2f	0xf3	0xfa	0xfa	0x04
0xcafe0160	0x32	0x37	0x73	0x11	0x42	0x08	0x00	0x80
0xcafe0168	0x00	0x01	0x00	0x02	0x00	0x04	0x00	0x08
0xcafe0170	0x42	0x23	0xc2	0xad	0xde	0x24	0x19	0x8f
⋮	⋮							
0xcafe0a30	0x20	0x44	0x70	0x7f	0x78	0x00	0x08	0x1a
0xcafe0a38	0x00	0x00	0xbe	0xbb	0x8e	0x0f	0x13	0xff
0xcafe0a40	0x04	0x0f	0xff	0xfe	0x02	0x17	0x22	0x04
0xcafe0a48	0x08	0x84	0x04	0x0f	0xff	0x10	0x02	0x70
0xcafe0a50	0x00	0x90	0xbe	0xba	0x8d	0x0f	0x15	0x4b
0xcafe0a58	0x00	0x90	0xef	0xbe	0x97	0xaf	0x88	0x3a
0xcafe0a60	0x24	0xff	0xff	0x08	0x04	0x05	0x06	0xf2
⋮	⋮							
0xfffe03a8	0x08	0x90	0xbe	0xba	0x7f	0x7c	0xf8	0x00
0xfffe03b0	0xb8	0x90	0xbe	0xba	0x7e	0x7d	0x07	0x05
0xfffe03b8	0x23	0x02	0xaa	0x02	0xdf	0xa3	0x2a	0x0f
0xfffe03c0	0x45	0x40	0x1a	0x00	0xde	0x39	0x02	0xf7
0xfffe03c8	0x67	0x0a	0xb5	0x04	0xf2	0x2d	0x42	0x77
0xfffe03d0	0x09	0x07	0x23	0x3d	0xda	0x97	0x34	0x81
0xfffe03d8	0xa0	0xb0	0x34	0x08	0x00	0x2a	0x27	0x23

3 Practical part: Running a command remotely (50pts)

For this Section, you are going to develop a client-server application that uses TCP/IP to communicate over the Internet. The task of the application is the following: instead of running a command locally, for example

```
$ date -u
Tue Nov 12 23:43:16 UTC 2024
```

the client program is going to connect to a server, specified by an Internet address (a FQDN) and a port number, and push the command to run as well as the different arguments to the command over to the server. The server will then execute the command remotely. Standard input of the client will be forwarded to the TCP/IP connection on the client side and will be forwarded on the server side to a file descriptor that will serve as the actual processes' standard input. Standard output of the process will be forwarded on the server side over the TCP/IP connection to the client, where it will be copied directly onto standard output. Nonwithstanding buffering effects, the application will run remotely but interactively.

For example, we can have a server running on the Internet address `heuschrecke.christoph-lauter.org`, listening for connections on port 9999. A client can then run a command remotely by executing:

```
$ ./client heuschrecke.christoph-lauter.org 9999 date -u
Tue Nov 12 23:49:24 UTC 2024
```

Here, `date` is the name of the executable that will be run by the server. The string `-u` is an argument. When started, the server will not know which command it will eventually run. It will receive all these pieces of information over the TCP/IP connection.

Your instructor has already entirely programmed out the client program, in the form of program `client.c`. That code is available for your reference on the class website.

The client program does the following:

- It verifies it received at least 3 arguments, excluding its own name. It considers argument 1 to be a server name, argument 2 to be a port name, argument 3 the command to run remotely and arguments 4 and subsequent to be the remote command's arguments.
- It also verifies that there are no more than $2^{16} - 1$ arguments to be transmitted to the server, including the command name. It verifies as well that no argument string (or the command string) is longer than $2^{16} - 1$ characters.
- The client opens a TCP/IP connection to the the server with the server and port names that have been given. If this connection is successful,
- it sends a 16bit unsigned integer in network byte order (big endian) to the server over the TCP/IP connection. That integer indicates the number of arguments (including the command) that will follow.
- It then sends per argument a 16bit unsigned integer indicating the size of the string that will follow and the argument string (without its end sentinel `'\0'`).
- It then starts sending every byte it receives on its standard input over the TCP/IP link and copies every byte it receives from the TCP/IP link onto standard output.
- It stops as soon as it receives a End-Of-File condition on standard input or on the TCP/IP link.

The server program must do the complementing part:

- It needs to verify it got started with a valid port indication in argument.
- It then needs to open a socket, bind it to that port and listen for incoming connections.
- As soon as it has an incoming client connection, it needs to accept that connection.
- It then needs to read the number of strings to follow as a 16bit unsigned integer in network byte order and allocate a correctly sized array for this vector of arguments.
- It then needs to read off the TCP/IP connection the length of each argument string, followed by the argument string itself. The memory space for the string needs to be properly allocated.
- It finally needs to fork off a child process that is properly connected to the TCP/IP connection, possibly with two pipes (one to the child, one from the child). The child needs to close all unneeded file descriptors, use `dup2` to replace its existing standard input and standard output by a file descriptor leading (directly or indirectly) to the TCP/IP connection and then must use `execvp` to replace its executable by the command's executable with the arguments received.
- The parent process needs to wait for the child to die, forwarding all data from the TCP/IP connection to the child and back from the child to the TCP/IP connection (unless another technique is used instead of pipes).
- At termination, all processes must close all file descriptors and deallocate all memory they have allocated.

No boilerplate code is provided for this `server.c` program. You find enough bits and pieces to use in the programs that we have written in class, in the previous homeworks and exams, as well as in the client code `client.c`. Make sure that your program does not leak memory and that it closes all file descriptors it can close. Remember to use `select` when you need to read on two file descriptors concurrently. The `client.c` code contains an example use of `select`.

Test your server program using the provided client program. For example, run

```
$ ./server 9999
```

in one terminal and then connect to the server over a local TCP/IP connection with

```
$ ./client localhost 9999 cat
```

The server should then run the `cat` program (in a child process) that repeats everything it sees on standard output on standard input. This means that on the client side you should see input and output repeating:

```
$ ./client localhost 9999 cat
Hello World
Hello World
How are you?
How are you?
```

The connection should drop when you press Ctrl-D to send End-Of-File.