CS4375 Operating Systems Concepts
Fall 2024. Final Exam
12/13/2024 – 10:00AM to 12:45PM MST (theoretical part)
12/13/2024 – 10:00AM to 11:59PM MST (practical part)

All documents allowed
Electronic devices **without** Internet connectivity allowed[1]
You need to turn in the theoretical part of the exam (Sections 1, 2, 3 and 4,) by 12/13/2024 12:45PM MST on paper, directly to your instructor. You need to turn in the practical part of the exam (Section 5) by 12/13/2024 11:59PM MST by email to `utep-os-fall-2024-final@christoph-lauter.org`.

1 Ingredients of an Operating System

1. Give a clear definition of...
   - what an executable is,
   - what a process is, and
   - what user space and kernel space are.
2. Give a list of operations a typical Unix/Linux file system supports. It is possible to list 13 operations or more.
3. Give a *yes/no* answer, along with an explanation, to the following questions:
   (a) In a multi-core system, there can only be one process running for a given executable.
   (b) Two processes generally access the same address space.
   (c) Two processes having shared access to part of a computer's memory see that shared part of the memory at the same virtual address.
   (d) Two threads can communicate through memory they share.
   (e) An executable, run two times consecutively, will result in two processes that see the exactly the same address space each time, with pointers having the same numerical value for each run.

2 Virtual and Physical Addresses

Assume a 32bit big-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[2]. Let the physical base address of the first page table be `0xbeef0000`.

[1] Put your device into airplane mode.
[2] Everything that starts in `0x` in this exercise is notated in hexadecimal (base 16).

Use the memory extract and the base address to translate the virtual address `0x54321abe` to a physical address. If you cannot perform this translation because the page is not mapped in, indicate that a page fault occurs. In your answer, detail each step of the translation; do not just give the final translation result.

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

Address (32bit)	*Content (64bit)*
⋮	⋮
0xbeef0a68	0xabad1dea00000002
0xbeef0a70	0xdecafbad00000007
0xbeef0a78	0x12340000beef0003
0xbeef0a80	0xdead0000affe0007
0xbeef0a88	0xcafe000055550004
0xbeef0a90	0xb15b00b500000000
⋮	⋮
0xcafe19e8	0x0011002200330044
0xcafe19f0	0xcafeb00000000008
0xcafe19f8	0xbadeaffe00000000
0xcafe1900	0xaabbccdd00000003
0xcafe1908	0xabad1dea66666666
0xcafe1910	0xffff000000000007
0xcafe1918	0xfffe000000000004
⋮	⋮
0xdead19f0	0x0000000000000000
0xdead19f8	0xbadf00d0decaf003
0xdead1900	0xabadbabe00700704
0xdead1908	0xcafeb000c0de0003
0xdead1910	0xdead0000face0000
0xdead1918	0x1234000077770009
0xdead1920	0xbadeaffe00000666
⋮	⋮

3 Memory Mappings After a Call to `fork()`

Suppose the following code is run on a 64bit Linux machine with 48Gbyte of main memory. The system uses 4096byte wide pages. Besides the processes spawned as a consequence of running the code, there are no other (memory hungry) processes run on the system. Explain how many page faults are expected to occur when the code executes lines 20 through 24 and why this is the case. Further explain how many page faults will occur during the execution of lines 27 through 29 and why this is the case. In your answer, precisely describe and refer to the strategy Linux uses for page handling after a call to `fork()`.

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

```
#include <sys/wait.h>

#define BUFFER_LENGTH    ((((size_t) 4096) * ((size_t) 1048576)))

int main(int argc, char **argv) {
  unsigned char *buf;
  size_t i;

  buf = (unsigned char *) malloc(BUFFER_LENGTH);
  if (buf == NULL) return -1;

  for (i=0;i<BUFFER_LENGTH;i++) {
    buf[i] = (unsigned char) i;
  }

  if (fork() == 0) {
    for (i=0;i<BUFFER_LENGTH;i++) {
      if (buf[i] == ((unsigned char) 'A')) {
        printf("%c", ((char) buf[i]));
      }
    }
    printf("\n");

    for (i=0;i<BUFFER_LENGTH;i++) {
      buf[i] = (unsigned char) (i + 1);
    }

    for (i=0;i<BUFFER_LENGTH;i++) {
      if (buf[i] == ((unsigned char) 'A')) {
        printf("%c", ((char) buf[i]));
      }
    }
    printf("\n");

    free(buf);
    return 0;
  }
  wait(NULL);

  free(buf);
  return 0;
}
```

4 POSIX File Systems

Indicate what the following program displays when it is run on a POSIX-compliant file system and why this is the case.

```
#include <string.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char **argv) {
  int fd1, fd2;
  char str1[] = "naughty";
  char str2[] = "nice";
  char buf1[42];
  char buf2[42];

  fd1 = open("file.txt", O_CREAT | O_RDWR, 0644);
  write(fd1, str1, sizeof(str1));
  lseek(fd1, 0, SEEK_SET);

  unlink("file.txt");

  fd2 = open("file.txt", O_CREAT | O_RDWR, 0644);
  write(fd2, str2, sizeof(str2));

  memset(buf1, '\0', sizeof(buf1));
  read(fd1, buf1, sizeof(buf1));

  close(fd1);

  fd1 = open("file.txt", O_CREAT | O_RDWR, 0644);

  unlink("file.txt");

  memset(buf2, '\0', sizeof(buf2));
  read(fd1, buf2, sizeof(buf2));

  close(fd2);
  close(fd1);

  printf("%s or %s\n", buf1, buf2);

  return 0;
}
```

5 Parallel Execution (Practical Part)

For this Section, you need to come up with a program `runparallel.c`, written in C, that does the following:

- The program starts by checking it got at least 3 arguments:
  - its own name,
  - a string indicating a positive integer n and
  - another string, indicating the name of another executable.

  If extra arguments are given, the program keeps them around, as they will be needed later.

  If not enough arguments are given, the program displays an error message on standard error and terminates with a failure status code. Similarly, if the argument that is supposed to be a positive integer n does not evaluate to such a positive integer, the program terminates with an error message and a failure status code.

- The program then forks off n children processes, numbered child 0 through child $n-1$. Each of the children processes does the following:
  - It converts its number i to a string. This means, the first child converts the integer 0 to a string, the second child the integer 1 and so on until the last child, which converts integer $n-1$ to a string.
  - The child then replaces its own executable by the executable whose name was given to its parent. It uses `execvp` for this replacement. As arguments, the new executable receives (1) its own name, (2) its number as a string and (3) all other extra arguments its parent had received.
  - In case any of the actions fails, an error message is displayed on standard error and the child exits, returning a failure status code.
  - All children stay connected to the same standard input, standard output and standard error that they inherited from their parent.

- After forking off the children, the parent process, which kept the childrens' process IDs, waits for each one of its children process to die. It uses `waitpid` for that task. The parent explicitely waits for each child, not just any process to terminate.

- If anything goes wrong, the parent tries to recover the situation but displays an error message on standard error. It does fail with a failure status code if no recovery is possible. In the case when at least one but not all children could be forked off, the parent needs to still wait for the children that did get forked off.

Both the parent and the child processes need to allocate memory on the heap. They can do so using `calloc`. They need to free all that allocated memory using `free` though.

No boilerplate code is provided for the programming exercise of this Section. You need to submit C source code in a file named `runparallel.c`. The code needs to compile on `dandelion` using

```
gcc -Wall -O3 -o runparallel runparallel.c
```

Below you find example runs of the `runparallel` program:

```
$ ./runparallel
Usage: ./runparallel <num of jobs> <executable> [<arg1> ... <argn>]
$ ./runparallel 0 echo
Cannot convert "0" to positive integer
$ ./runparallel hello echo
Cannot convert "hello" to positive integer
$ ./runparallel 3 echo Hello World
0 Hello World
1 Hello World
2 Hello World
$ ./runparallel 10 echo Hello World
0 Hello World
1 Hello World
3 Hello World
4 Hello World
6 Hello World
2 Hello World
5 Hello World
9 Hello World
7 Hello World
8 Hello World
$ ./runparallel 10 sleep
```

– Good luck! – ¡Buena suerte! –