CompSys👏 Review👏

Understanding the OS, Races, VM Address Translation

TA: Julian Philippe Pedersen `<jupe@di.ku.dk>`

CompSys 2022-2023

Understanding the OS

Kernel

- Kernel is a protected part of the OS.
- Only the kernel has direct access to:
  - Hardware
  - System memory
  - File I/O
- System calls require access to the kernel.

Programs vs. Processes

- A program is just a list of instructions for the computer. (A file containing code)
- A process is an instance of a program. This instance is “live” and stateful.
  - When people say “program,” they often mean “process.”
- When multiple processes are running, the OS utilizes context switching to give the illusion of concurrency.
  - Context switching requires the saving of the state of a process. To do this, it must:
    - All registers (including control registers)
    - Contents of memory

Races

- Race conditions: a situation where values depend on order of execution of instructions
  - Problem in concurrent execution
  - Multithreaded solutions: semaphores (mutex), conditional variables
  - Less of a problem with threads (but race conditions can still exist!)

Threads

- Run in the same address space as the calling process
- Threads have their own thread context, which includes:
  - Thread ID
  - Stack + Stack pointer
  - Program counter
  - General-purpose registers
  - Condition codes

Dealing with Thread Concurrency

- Semaphores (mutexes)
  - Invariant: must always be non-negative
    - `P() // Prolaag, ”try”. Decrements semaphore, unless decrement would cause semaphore to become < 0.`
    - `V() // Verhoog, ”increment”. Increments semaphore.`
- Conditional variables can allow blocking and signaling specific threads
  - Watch out for <u>spurious wakeups</u>! (When a thread continues execution even when the valid condition is not true.)
  - When in doubt, <u>while()</u> it out!
    - This ensures that when a thread is signaled to wake up, it will always check if the condition for it to continue is true.
    - If-statements cause this check to only occur once during a thread’s lifetime. This is not necessarily incorrect, but it can be difficult to guard against spurious wakeups.

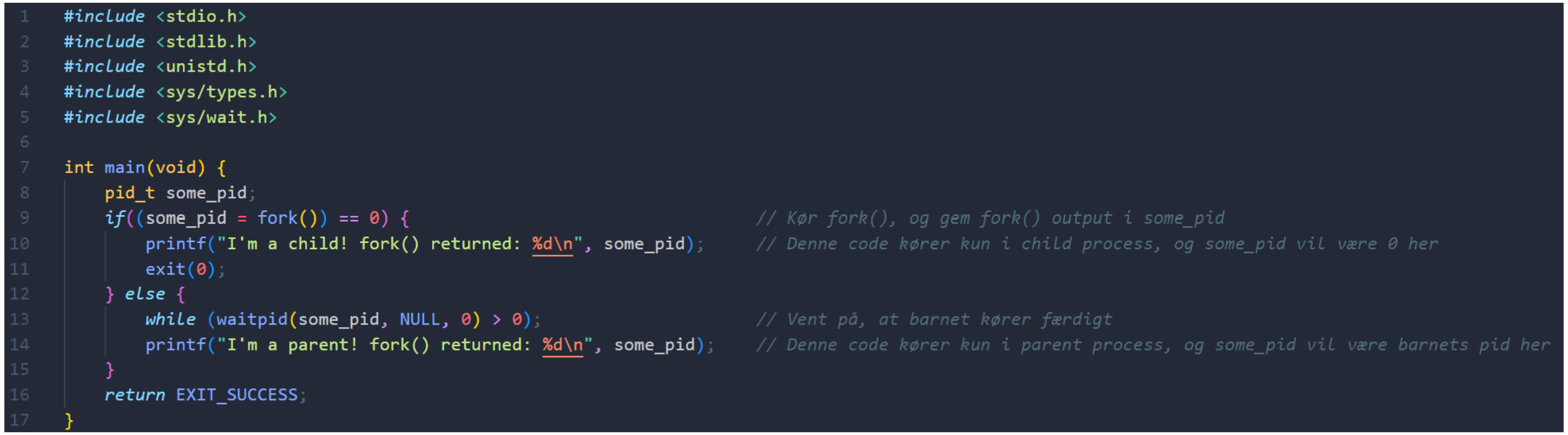
Processes

- Processes contain a duplicate of the parent's information, but in a separate address space.
  - Changes made in one process are <u>not</u> reflected in the others!
- Child processes must be <u>reaped</u> by their parent.
  - Processes not reaped by the parent upon parent termination will be "adopted" by the `init` process (pid 1).
  - Long-running child processes adopted by `init` that never terminate are called "zombie" processes.

Fork()

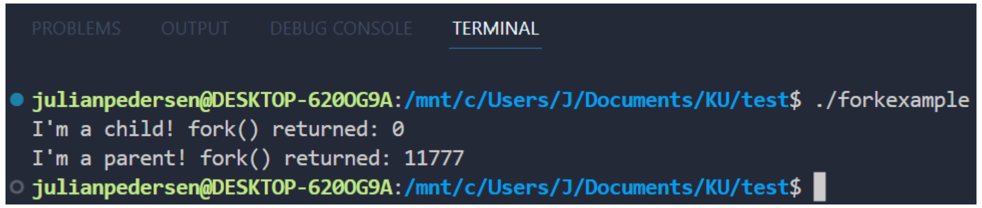
- `Fork()` returns two values
  - Child's process id (PID) in parent process
  - 0 in child process
  - Use `fork()` return value to check whether the current program is a parent or child process
    - `fork() == 0`
      - `True`: We are in the parent process
      - `False`: We are in the child process
- `waitpid(pid, *status, options)` is used to block execution in parent processes based on child processes
  - `(waitpid(child_pid, status, 0) > 0)` blocks until child with `child_pid` is finished executing

Fork() cont.



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

int main(void) {
    pid_t some_pid;
    if((some_pid = fork()) == 0) {                                  // Kør fork(), og gem fork() output i some_pid
        printf("I'm a child! fork() returned: %d\n", some_pid);     // Denne code kører kun i child process, og some_pid vil være 0 her
        exit(0);
    } else {
        while (waitpid(some_pid, NULL, 0) > 0);                     // Vent på, at barnet kører færdigt
        printf("I'm a parent! fork() returned: %d\n", some_pid);    // Denne code kører kun i parent process, og some_pid vil være barnets pid her
    }
    return EXIT_SUCCESS;
}
```



```
julianpedersen@DESKTOP-6200G9A:/mnt/c/Users/J/Documents/KU/test$ ./forkexample
I'm a child! fork() returned: 0
I'm a parent! fork() returned: 11777
julianpedersen@DESKTOP-6200G9A:/mnt/c/Users/J/Documents/KU/test$
```

Process Graphs

- Very helpful in determining execution order of forked processes!
- Things to keep in mind:
  - `fork()` spawns two edges
  - `wait()` can "join" multiple edges into a single one (watch for child execution order!)
  - `exit()` causes the calling process to exit immediately. It will not continue execution of commands below.

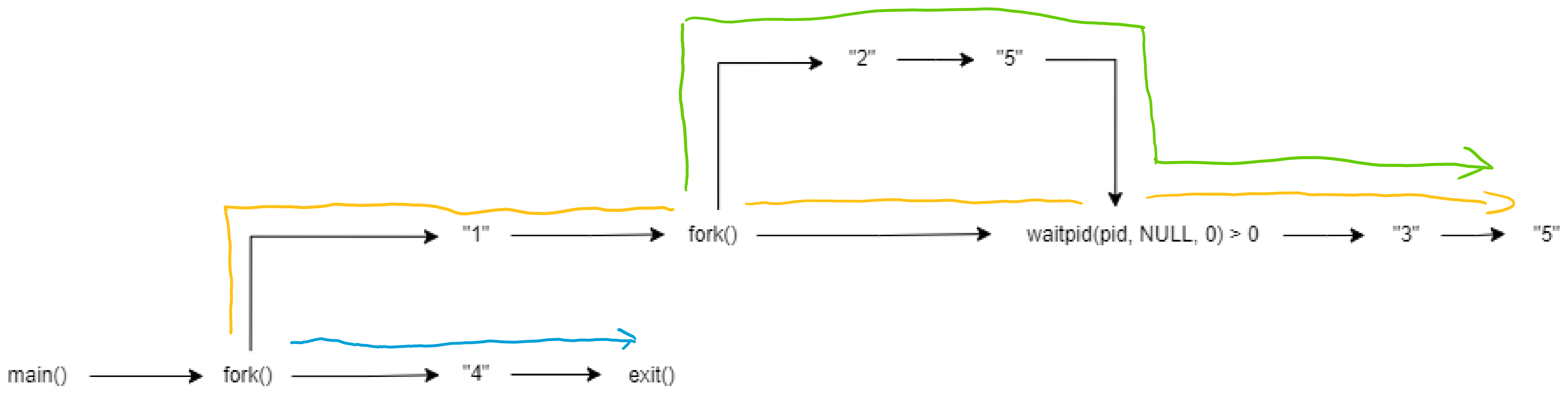
Fork() Example

```c
int main () {
  if (fork() == 0) {
    printf("1");
    if (fork() == 0) {
      printf("2");
    } else {
      pid_t pid; int status;
      if ((pid = waitpid(pid, NULL, 0)) > 0) {
          printf("3");
      }
    }
  } else {
    printf("4");
    exit(0);
  }
  printf("5");
}
```

```c
int main () {
  if (fork() == 0) {                              - Child
    printf("1");
    if (fork() == 0) {                            - Child-Child
      printf("2");
    } else {
      pid_t pid; int status;
      if ((pid = waitpid(pid, NULL, 0)) > 0) {
          printf("3");                            - wait until child-child
      }                                             has finished executing!
    }
  } else {
    printf("4");                                  - parent
    exit(0);                                      - Exit() stops parent here!
  }
  printf("5");                                    - Only child and child-child
}                                                   reach this line!
```

Fork() Process Graph

```
int main () {
  if (fork() == 0) {
    printf("1");
    if (fork() == 0) {
      printf("2");
    } else {
      pid_t pid; int status;
      if ((pid = waitpid(pid, NULL, 0)) > 0) {
          printf("3");
      }
    }
  } else {
    printf("4");
    exit(0);
  }
  printf("5");
}
```



VM Address Translation

Background

- VPA (Virtual Page Address)
  - VPA can be split into Virtual Page Number (VPN) and Virtual Page Offset (VPO)
  - VPN can be further split into Virtual Page Tag (VPT) and Virtual Page Index (VPI)
- PPA (Physical Page Address)
  - PPA can be split into Physical Page Number (PPN) and Physical Page Offset (PPO)
- TLB (Translation Lookaside Buffer)
  - Contains previously-accessed VPAs/PPAs by storing VPTs, PPNs, valid bits in “ways”
  - Analogous to cache
- Page Table
  - Contains Page Table Entries (PTEs)
  - PTEs consist of VPNs, PPNs, valid bits

Tips

- Read the description!
  - ☆ Look for keywords **page size**, **# sets**, **address format**
    - # lower-order bits = log2(page size)
    - # set index bits = log2(# sets)
    - Address format is almost always hexadecimal, but make sure to check! (do not confuse yourself between hexadecimal and base-10!)
- When you have found the number of bits for VPT/VPI/VPO/ and PPN/PPO, mark the ranges down on the test!
  - Easy to mess this up, so the less data in your “working set” (short-term memory) the better
- Each hexadecimal digit corresponds to 4 binary bits in the same order.
  - `0x00 = 0000 0000`, `0x01 = 0000 0001` … `0x10 = 0001 0000` … `0xff = 1111 1111`
  - Useful for quick translation between binary and hexadecimal

"Algorithm"

- Read description. Find **page size** and **# of sets**.
  - Determine number of bits for (V/P)PO with log2(page size) and TLBI with log2(# sets).
- Translate the given address from hex to binary and write the bits in the top bitfield.
- Translate **VPN**, **TLB index** (TLBI), **TLB tag** (TLBT).
- Lookup **TLB** with **TLBI** and **TLBT**
  - If (tag exists in index) && (valid bit = 1), **TLB hit** ☺
    - Copy **VPO** to **PPO** directly. Copy **PPN** from **TLB** to PPN field, right-to-left. You're done! ✓
  - If (tag does not exist in index) || (valid bit = 0), **TLB miss** ☹ Continue to next step.
- Lookup **page table** with **VPN**
  - If (VPN exists in page table) && (valid bit = 1), **no page fault**! ☺
    - Copy **VPO** to **PPO** directly. Copy **PPN** from **PTE** to PPN field, right-to-left. You're done! ✓
  - If (VPN exists in page table) || (valid bit = 0), **page fault** ☹ Continue to next step
- If (no TLB hit) && (page fault)
  - No valid PPN exists. Leave bottom bitfield empty. You're done! ✓

Example: TLB Hit

Bit position	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VA = 0x03d4	0	0	0	0	1	1	1	1	0	1	0	1	0	0
	0x0		0x3				0xd				0x4			

Each hex cipher corresponds to 4 bits in binary.

	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	TLBT						TLBI							
	0x03						0x3							
	0x0		0x3				0x3							
Bit position	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VA = 0x03d4	0	0	0	0	1	1	1	1	0	1	0	1	0	0
	0x0				0xf				0x1		0x4			
	0x0f								0x14					
	VPN								VPO					

1. Extracting VPN, TLBT, TLBI, VPO from a VA.

	11	10	9	8	7	6	5	4	3	2	1	0
	0xA				0x6				0x4			
Bit position	11	10	9	8	7	6	5	4	3	2	1	0
PA = 0xA64	1	0	1	0	0	1	0	1	0	1	0	0
	0x2		0x9				0x1		0x4			
	0x29						0x14					
	PPN						PPO					

3. Constructing PA from PPA and PPO.

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	0x03	—	0	0x09	0x0D	1	0x00	—	0	0x07	0x02	1
1	0x03	0x2D	1	0x02	—	0	0x04	—	0	0x0A	—	0
2	0x02	—	0	0x08	—	0	0x06	—	0	0x03	—	0
3	0x07	—	0	0x03	0x29	1	0x0A	0x34	1	0x02	—	0

HIT!

2. Looking up `0x03` in a TLB.