B4B35OSY: Operační systémy

Grafika, GUI a HW akcelerace

Michal Sojka[1]

20. ledna 2021

[1]michal.sojka@cvut.cz

Obsah I

Uživatelské rozhraní

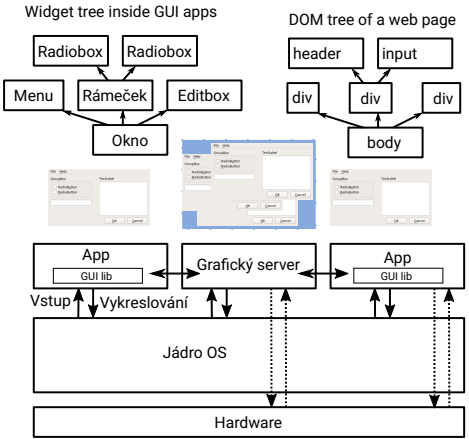
- Příkazová řádka (shell, ...)
- Textové rozhraní (konzolové aplikace – vim, top, Midnight Commander)
- Grafické rozhraní (grafická okna)
  - MS Word/Libre Office, Webový prohlížeč
  - V dnešní době je velká část aplikací postavena na webových technologiích. Tyto aplikace nepoužívají pro zobrazování grafiky přímo služby OS, ale služby webového prohlížeče, který pak využívá služby OS.

Cíl přednášky

- Podívat se „pod pokličku“ frameworkům pro tvorbu uživatelského rozhraní a pochopit jak interagují s jádrem OS a jinými komponentami.
- Naznačit, jak fungují GPU.

Koncept GUI z pohledu OS

GUI = graphical user interface



- Aplikace se typicky stará pouze o „své“ okno
- Kombinování oken různých aplikací má na starosti tzv. „grafický server“.
- Aplikace čte informace (události) o vstupu od uživatele
  - myš, klávesnice, touch screen
  - to má na starosti tzv. **smyčka událostí** (viz dále)
- Aplikace organizuje grafické prvky ve stromové struktuře
  - Události posílá do správných „objektů“ v aplikaci (např. aktuálně vybraný „edit box“)
  - „Objekty“ na události reagují a vykreslují se
- Vykreslování – čáry, text, obrázky, …
  - softwarové – do framebufferu v paměti, implementováno knihovnou, kromě finálního zobrazení není potřeba OS
  - HW akcelerované – příkazy k vykreslování se posílají do GPU (zpravidla za pomoci jádra OS)

Smyčka událostí

Event loop

- Typicky v hlavním vlákně aplikace:

```
while (!quit) {
  wait_for_event // keyboard, mouse, timer, ...
  handle_event   // send to focused widget for handling
}
```

- Hlavní smyčku většinou nepíše programátor, ale nachází se uvnitř GUI frameworku (např. `app.run()`)
- U vícevláknových aplikací může být smyčka událostí ve více vláknech
  - Hlavní vlákno řeší například události od GUI
  - Ostatní vlákna zpracovávají události zaslané hlavním vláknem (viz níže) nebo třeba události ze sítě
- Pokud obsluha události v hlavním GUI vlákně trvá dlouho, aplikace nemůže reagovat na jiné události
  - Projevuje se to jako „zatuhlá“ aplikace, či hláška „aplikace neodpovídá“

Dlouhotrvající obsluha události

- Déletrvající obsluha by se měla vykonávat mimo hlavní vlákno (v tzv. pracovním vlákně), aby neblokovala smyčku událostí.
- Z hlavního vlákna se pouze spustí – např. pomocí semaforu nebo zasláním události do smyčky v pracovním vlákně.
- Mnoho knihoven má omezení, že některé operace (např. vykreslování) lze volat jen z jednoho vlákna (jedná se o tzv. nereentrantní funkce)
- Proto je dokončení obsluhy v pracovním vlákně často signalizováno zasláním události zpět do hlavního vlákna (např. pomocí pipe či fronty zpráv). A teprve v hlavním vlákně se uživateli vykreslí/napíše že operace byla dokončena.

Neblokující I/O

Linux/Unix

- Základním problémem, který musí smyčka událostí řešit je čekání na více zdrojů událostí současně (myš, klávesnice, případně síť, …)
- Pokud aplikace otevře např. `/dev/input/mice` a zavolá `read()`, **vlákno se zablokuje** do té doby než uživatel pohne myší a na jiné vstupy (např. klávesnice) nemůže reagovat
- Je potřeba používat tzv. **neblokující I/O**
  - V Linuxu můžeme „file descriptor“ `fd` přepnout do neblokujícího režimu následovně:

```
/* set O_NONBLOCK on fd */
int flags = fcntl(fd, F_GETFL, 0);
fcntl(fd, F_SETFL, flags | O_NONBLOCK);
```

  - Pokud na neblokujícím `fd` nejsou žádná data ke čtení, `read()` se okamžitě vrátí s chybou `EAGAIN` nebo `EWOULDBLOCK`.

Neblokující I/O – pokračování

Linux/Unix

- U neblokujícího I/O ale nechceme pořád kontrolovat, jestli na některém „file descriptoru“ (FD) nejsou připravena data ke čtení (tzv. busy waiting), protože by to zbytečně zatěžovalo procesor
- OS poskytuje systémová volání, které umí čekat na více FD najednou – např.: `select()`, `poll()`, `epoll_wait()`
  - Všechna dělají v principu to samé, ale mají jiné API a různou výkonnost.
  - Aplikace řekne na co všechno chce čekat a funkce pak čeká.

poll() example

```c
int retval;
struct pollfd poll_list[3] = {
    // specify which socket and events we are interested in
    { .fd = mouse_fd; .events = POLLIN },
    { .fd = kbd_fd;   .events = POLLIN },
    { .fd = sock_fd;  .events = POLLIN | POLLOUT },
};
while(1) {
    retval = poll(poll_list, 3, 1000); // Wait for 3 FDs with 1000 ms timeout
    if (retval < 0) err(1, "poll"); // Print error message and exit
    else if (retval == 0) printf("timeout\n");
    else {
        if (poll_list[0].revents != 0) { /* read mouse_fd and process the data */ }
        if (poll_list[1].revents != 0) { /* read kbd_fd and process the data */ }
        if (poll_list[2].revents != 0) {
            if (poll_list[2].revents & POLLIN)  { /* read data from the socket and process them */ }
            if (poll_list[2].revents & POLLOUT) { /* socket is ready to send data - we can do it know */ }
        }
    }
}
```

Windows message loop

https://docs.microsoft.com/en-us/windows/desktop/winmsg/using-messages-and-message-queues

```
MSG msg;

while( (bRet = GetMessage( &msg, NULL, 0, 0 )) != 0) {
    if (bRet == -1) {
        // handle the error and possibly exit
    }
    else {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

- Windows poskytují vyšší úroveň – nízkoúrovňové události jsou převáděny na zprávy
- File descritpry a pod. jsou schovány uvnitř `GetMessage`
- Princip je podobný

GUI framework

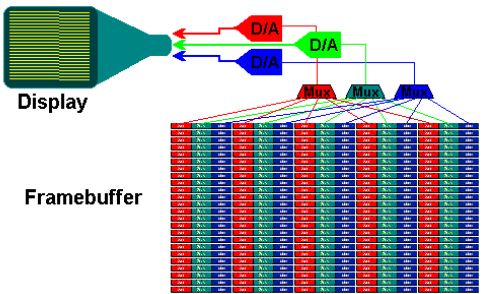
- Knihovny poskytující
  - Nezávislost na OS/HW
  - Vysokoúrovňové API (objekty, snadnost použití)
  - Základem každé knihovny je smyčka událostí
- WinForms – C#
- Qt – C++, různé OS i embedded HW
- GTK – C + podpora (bindings) jiných jazyků
- …

Grafický subsystém OS

- Dlouhá historie
- Technologie se rychle mění
- $\Rightarrow$ komplikované
  - různá API, zpětná kompatibilita, …

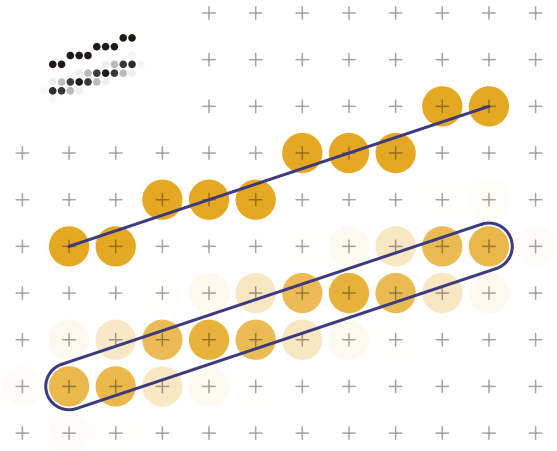
Framebuffer (obrazový buffer)

- Starší grafické karty implementovaly pouze tzv. framebuffer
- Framebuffer je paměť jejíž obsah graf. karta převádí na signál pro displej (např. VGA)
  - dedikovaná paměť na kartě či sdílená paměť s CPU
- „Surface“ v dnešních GPU je kus paměti reprezentující např. jen jedno okno
- Vykreslování = zápis do paměti (SW)
- Dnes: low-end embedded systémy



- Formát pixelů
  - 888 RGB, 888 RGBA, 565RGB 5551RGBA, YUV
  - ABGR 8888
  - dříve se používalo indexování (8b.) do palety

Příklad rasterizace čáry do framebufferu



Source: Wikipedia, Phrood, CC-BY-SA 3.0

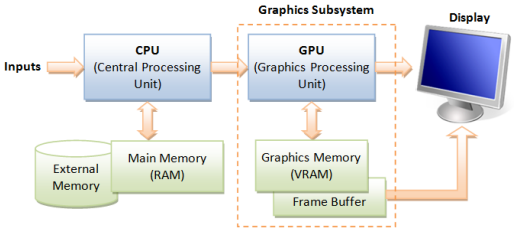
HW akcelerace

- SW vykreslování je pomalé
  - zejména při vysokých rozlišeních
  - skládání výsledného obrazu vyžaduje mnoho „kopírování“
  - poloprůhlednost objektů vyžaduje mnoho stejných výpočtů – např. výsledná barva (pixel p) je průměrem barev pixelů p_1 a p_2 dvou objektů: $p = 0.5p_1 + 0.5p_2$
  - Anti-aliasing, …
- Dnešní GPU je velmi výkonný paralelní počítač, který mnoho operací urychlí, nebo kompletně vykoná místo hlavního CPU
- Historicky se HW akcelerace vyvíjela:
  - 2D akcelerace
  - Pevná 3D akcelerace
  - Programovatelná 3D akcelerace
  - ...

2D akcelerace

- HW vykreslování kurzoru myši
  - Jeden z nejstarších typů akcelerace
  - Aby kurzor neblikal při posunování
  - Omezená velikost (kus paměti vyhrazen pro kurzor)
  - Implementováno v poslední fázi zpracování obrazu (tzv. scanout), kdy se pixely posílají na obrazovku
- Blitter – akcelerace operací s obdélníky
  - Kopírování obdélníků
  - Výplň obdélníku konstantní barvou
  - Kopírování se roztažením/smrštěním
  - Kopírování s průhledností [Potter/Duff operátory]
- Overlay (bluescreen)
  - GPU vkládá video do místa, kde aplikace „nakreslí modrý obdélník“
- Dříve měly GPU samostatný HW pro 2D, dnes je univerzální programovatelný HW, který umí 2D i 3D
- Dnes tento typ akcelerace nabízí i malé mikrokontroléry (čipy za pár korun)

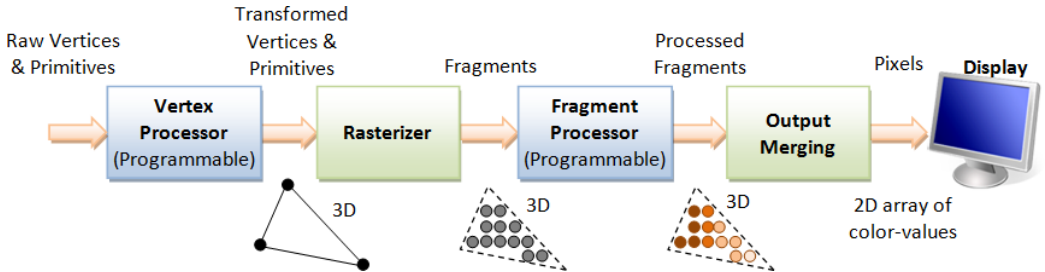
3D akcelerace



- Dvojité bufferování = vykreslování následujícího obrázku do „neviditelné paměti“ zatímco displej zobrazuje předchozí (hotový obrázek)

- Aplikace nezapisuje přímo do framebufferu
- Posílá příkazy a data do GPU
- GPU provádí vykreslování (zápisy do paměti) samo a paralelně na mnoha procesorech
- Výsledky ukládá buď přímo do framebufferu nebo do „neviditelné“ paměti, která je přístupná aplikacím jako „surface“

3D pipeline



- Vstupem je pole „vertexů“ (x, y, z) + další informace
- Vertex procesor pracuje s vektory (rotace, posun, …)
- Fragment procesor „obarvuje“ (stínování, textury)
- Výstupem je rastrový obrázek
- Pro práci GPU využívá mnoho paměťových oblastí
  - vstupní vertexy, výstupní rastr, hloubkovou mapu (z-buffer), …

Komunikace s GPU z pohledu OS

1. Paměťově přístupné I/O (MMIO)
   - Registry
   - Část paměti na GPU
2. DMA
   - Kopírování dat
   - Fronta příkazů
3. Přerušení
   - dokončení operace

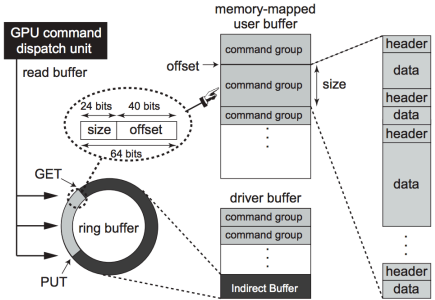


Figure 4. GPU command submissions.

- Příkazy pro GPU si připraví aplikace ve spolupráci s knihovnami (např. libGL.so)
- Knihovna je závislá na HW
- Ovladači GPU (v jádru OS) se předá ukazatel na příkazy pro GPU
  - Ovladač provede bezpečnostní a jiné kontroly a vloží novou položku do kruhového bufferu
- GPU čte příkazy z kruhového bufferu, vykonává je a výsledky zapisuje do paměti

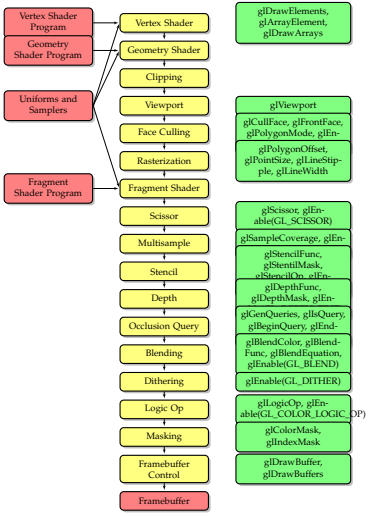
DRI/DRM (Linux)

- Direct Rendering Interface/Direct Rendering Manager
- Dovoluje více aplikacím současný přímý přístup ke GPU (skrze knihovny)
- Obsahuje paměťový alokátor pro paměť na GPU
- Řeší koherenci paměti mezi CPU–GPU
  - Nutnost explicitně vyprázdnit cache po dokončení operací apod.

(De)kódování videa

- Probíhá v několika fázích
- Dekódování
  1. Dekomprese („unzip“)
  2. Inverzní diskrétní kosinová transformace
  3. Kompenzace pohybu
  4. Převod barevného prostoru (YUV RGB)
  5. Zvětšování/zmenšování

OpenGL



- Specifikace definující platformě nezávislé API pro grafické operace
- Lze implementovat pomocí SW renderování nebo HW renderování
- Vykreslovací pipeline je složitější než na předchozím obrázku

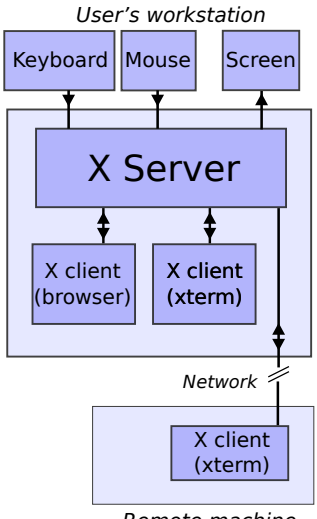
Zdroj: Stéphane Marchesin, Linux Graphics Drivers: an Introduction

Grafické servery

- X server (Unix)
- Kompozitory

X server – grafický server pro UNIX

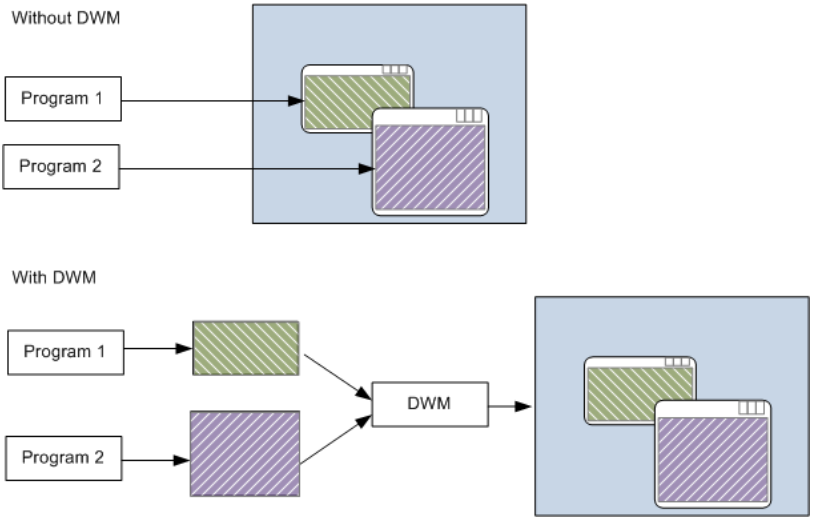
- Privilegovaná aplikace, umožňující ostatním aplikacím grafický výstup a čtení událostí
- Komunikace pomocí protokolu (socket)
- Síťová transparentnost
- Mimo jiné implementuje i schránku (clipboard) apod.
- Dnes
  - Aplikace nevykreslují pomocí komunikace s X serverem, ale pomocí komunikace s GPU
  - Dnes je X server „jen“ obálka implementující clipboard a kombinující okna aplikací dohromady (nadsázka)



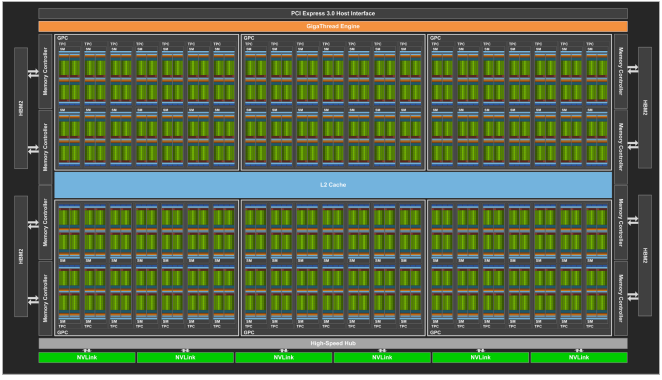
Grafický kompozitor

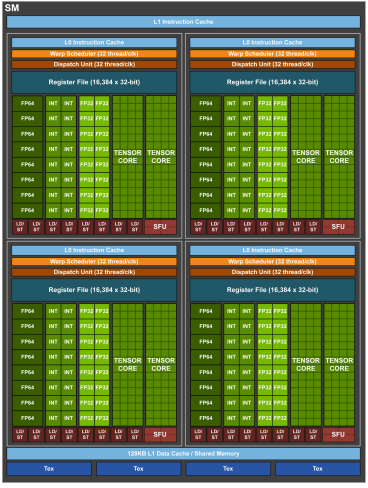
- V Linuxu např. Wayland, ve Windows od Windows Vista (DWM)
- Aplikace renderují své okénko samy pomocí přímé komunikace s GPU
- Výsledný „surface“ předají do kompozitoru
- Kompozitor přidá rámečky, stíny, průhlednost, animace, atd. a vytvoří výslednou podobu celé obrazovky (rovněž pomocí GPU)
- Při komunikaci mezi aplikací a kompozitorem se nemusí „surface“ kopírovat – „surface“ spravuje jádro a aplikace si předávají se jen odkazy (např. file descriptor)
  - Musí být zajištěna bezpečnost, aby neoprávněné aplikace nemohly vidět/modifikovat okna jiných aplikací.

Grafický kompozitor



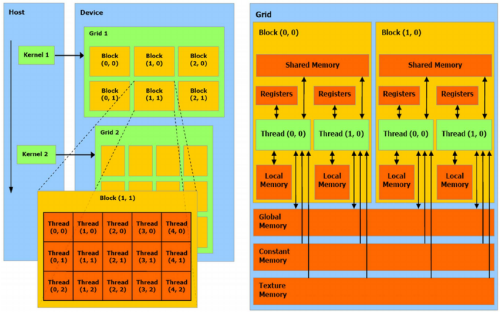
HW architektura dnešních GPU





Paměťová hierarchie

- Různé adresové prostory
- MMU jednotka není jen v CPU, ale i v GPU
  - Adresy: CPU virtuální, CPU fyzické, GPU virtuální, GPU fyzické
- GPU paměť není koherentní s CPU



Psaní paralelních programů

- Většinou jako rozšíření jazyků C/C++
- Pro Akcelerátory
  - CUDA
  - OpenMP
  - OpenCL
- Pro multi-core CPU
  - OpenMP
  - TBB
  - Cilk Plus

CUDA

- Nízkoúrovňové API od firmy NVIDIA
- Kernel – funkce, která je vykonávána paralelně na akcelerátoru

```
#include <stdio.h>

__global__
void saxpy(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int main(void)
{
  int N = 1<<20;
  float *x, *y, *d_x, *d_y;
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));

  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));

  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

  // Perform SAXPY on 1M elements
  saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);

  cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = max(maxError, abs(y[i]-4.0f));
  printf("Max error: %f\n", maxError);

  cudaFree(d_x);
  cudaFree(d_y);
  free(x);
  free(y);
}
```

OpenMP

- Původně pro vyvíjeno paralelní CPU (multi-core)
- Později přidána i podpora speciálních akcelerátorů jako GPU pomocí *#pragma omp target*...
- Anotace pomocí direktiv

```
int main(int argc, char **argv)
{
  int a[100000];

  #pragma omp parallel for
  for (int i = 0; i < 100000; i++) {
    a[i] = 2 * i;
  }

  return 0;
}
```

OpenCL

```
// Multiplies A*x, leaving the result in y.
// A is a row-major matrix, meaning the (i,j) element is at A[i*ncols+j].
__kernel void matvec(__global const float *A, __global const float *x,
                     uint ncols, __global float *y)
{
  size_t i = get_global_id(0);              // Global id, used as the row index.
  __global float const *a = &A[i*ncols];    // Pointer to the i'th row.
  float sum = 0.f;                          // Accumulator for dot product.
  for (size_t j = 0; j < ncols; j++) {
    sum += a[j] * x[j];
  }
  y[i] = sum;
}
```