Recap: Concurrency Control

Alexander Christensen

Department of Computer Science
University of Copenhagen

2019

Overview

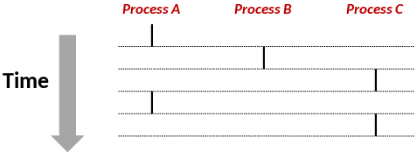
1. What is concurrency?
2. How can we achieve concurrency?
   - Processes
3. What can go wrong in concurrent programs

What is concurrency?

"Logical control flows are *concurrent* if they overlap in time." [2]

Concurrent Processes

- Each process is a logical control flow.
- Two processes *run concurrently* (*are concurrent*) if their flows overlap in time
- Otherwise, they are *sequential*
- Examples (running on single core):
  - Concurrent: A & B, A & C
  - Sequential: B & C



[2] (BOH, cp. 12)

How can we achieve concurrency?

We have 3 different set of tools available to use for writing concurrent programs:

Processes

- A running program can clone itself into one or more child processes
- Completely separated memory areas

I/O Multiplexing

- Resembles event-driven programming
- I/O operations do not block

Threading

- Multiple parallel control flows exist within the same process
- Often the most practical solution

Processes

How do we use them?

Creating Processes

- *Parent process* creates a new running *child process* by calling `fork`

- `int fork(void)`
  - Returns 0 to the child process, child's PID to parent process
  - Child is *almost* identical to parent:
    - Child get an identical (but separate) copy of the parent's virtual address space.
    - Child gets identical copies of the parent's open file descriptors
    - Child has a different PID than the parent

- `fork` is interesting (and often confusing) because it is called *once* but returns *twice*

Processes

Exam question from re-exam 19/20:

```
int main() {
  int status;

  printf("Hello ");
  fflush(stdout);
  printf("%d ", !fork());

  if (wait(&status) != -1)
    printf("%d ", WEXITSTATUS(status));

  printf("Bye ");

  exit(2);
}
```

Which of the following are *possible* valid outputs of the program?

a) `Hello 0 1 Bye 2 Bye`
b) `Hello Bye 1 0 2 Bye`
c) `Hello 1 0 Bye 2 Bye`
d) `Hello 1 Bye 0 2 Bye`
e) `Hello 0 Bye 1 2 Bye`
f) `Hello 0 1 Bye Bye 2`

Texture mapping

OpenGL performs texture mapping automatically, based on the chosen filtering function, and the *interpolation quilifiers* that we specify in our fragment shader:

```
in vec2 texCoord; // default : perspective-correct interpolation

smooth in vec2 texCoord; // the same

noperspective in vec2 texCoord; // smooth, but not perspective-correct

flat in vec2 texCoord; // no interpolation
```

Reading an image from disk

Assume a 2D texture with four color channels (RGBA) stored as a file "ourImage.png":

```
int width, height;
GLubyte* ourTextureData = someFunctionToReadFromDisk("ourImage.png",
    &width, &height);

GLuint ourTexture;
glGenTextures(1, &ourTexture);
glBindTexture(GL_TEXTURE_2D, ourTexture);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA,
    GL_UNSIGNED_BYTE, ourTextureData);

// remember to cleanup once the image is buffered to the GPU
delete ourTextureData;
```

What is a Texture - Binding the texture

To use our texture, it must be bound within the context of an activated shader program:

```
// activate our shader as the current program in
// OpenGL's state machine
GLuint ourShader = SomeFunctionToCreateAShader();
glUseProgram(ourShader);
```

Now we need to bind the texture to texture unit #0:

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, ourTexture);
```

Maximum number of texture units (simultaneously bound textures) can be queried:

```
int maxTextureUnits = 0;
glGetIntegerv(GL_MAX_TEXTURE_UNITS, &maxTextureUnits);
```

Using the texture in a rendering call

When the texture is bound we tell the shader where to find it:

```
GLint location = glGetUniformLocation(ourShader,
    "nameInFragmentShader");
glUniform1i(location, 0); // 0 corresponds to the chosen texture unit
```

Now the texture is bound within the shader, and we have buffered its position on the GPU as well. Now we can use our rendering commands to render the texture onto our surface.

Texture Mapping - Vertex Shader

The texture coordinate should be buffered together with the vertex position, as a per-vertex attribute:

```
#version 330 core

layout (location = 0) in vec2 vertexPos;
layout (location = 1) in vec2 texCoord;

out vec2 interpolatedTexCoord;

void main()
{
    gl_Position = vec4(vertexPos, 0.0f, 1.0f);
    interpolatedTexcoord = texCoord; // will be interpolated by OpenGL
}
```

Example with 2 textures

Render two triangles, but this time buffer two textures and perform various smooth transitioning effects between them based on the cursor position on the screen.

Framebuffer Example: Conway's Game of Life

Idea:
Use a framebuffer with 2 textures attached, use one for reading, the other for writing.

Algorithm:
Bind the framebuffer. Make a drawing call to read from `read` texture and write to `write` texture. Unbind the framebuffer. Make a drawing call again with `write` texture as uniform. Swap `read` and `write`.

The bigger perspective - Inspiration

Many common 3D rendering techniques use textures:

- normal mapping
- bump mapping
- parallax mapping
- static illumination

Many common 3D rendering techniques rely on framebuffers:

- deferred rendering
- shadow mapping
- screen-space ambient occlusion (SSAO)
- fast approximate anti-aliasing (FXAA)

Summary

We have seen the texture coordinate system.

We have looked at techniques for filtering, clamping, and wrapping of textures.

We have seen how textures can be used to enhance visual effects.

Some code examples have shown how textures can be used in OpenGL 3.3.