Recap: Concurrency Control

Alexander Christensen

Department of Computer Science
University of Copenhagen

2021

Overview

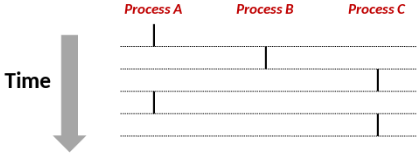
1. What is concurrency?
2. How can we achieve concurrency?
   - Processes
   - I/O Multiplexing
   - Threading
3. What can go wrong in concurrent programming
4. Concurrency protection mechanisms

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*

How to use Processes

Linux interface for using processes: `fork`, `execv`, `getpid`, `wait`, `waitpid`, `exit`.

Concurrency protection mechanisms are not necessary between a process and its child processes because they operate on completely disjoint memory areas.

However, communication between processes is expensive. Common methods are memory mappings (`mmap()`), file pipes (`pipe()`), and sockets.

Processes: Exam question example

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`

I/O Multiplexing

I/O operations are blocking. In a webserver `accept()` blocks control flow until it receives an incoming connection request.

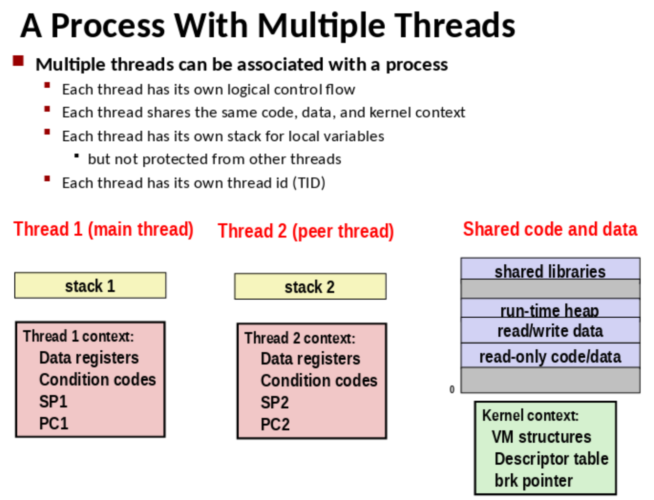
But with I/O Multiplexing we can add a number of file descriptors to a read set, e.g. `read-set := {STDIN_FILENO, listenfd}`. This way, a webserver can read from both without blocking, whenever either is ready for reading.

Programming interface: `select`, `FD_ZERO`, `FD_CLR`, `FD_SET`, `FD_ISSET`.

Detail: `select()` is a blocking function.
Example in (BOH, sec. 12.2).

Threading

Using multiple threads (ie. "multi-threading") is the most common way of achieving concurrency in programming.



How to use Threads

Linux (posix) programming interface: `pthread_create`, `pthread_join`, `pthread_detach`, `pthread_cancel`, `pthread_exit`, `pthread_self`, `pthread_yield`.

Threads are advantageous in many situations because sharing data btw. peer threads is very easy (`malloc()`), and overhead for creating threads is significantly less than for creating processes. Also, compared to I/O Multiplexing, threads are **truly** parallel.

Threaded programs are exposed to a lot of nasty bugs that can be difficult and time-consuming to track down.

Data sharing in a threaded program

Both global and dynamic (malloc'ed) data is shared between threads:

Example Program to Illustrate Sharing

```c
char **ptr;  /* global var */

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    Pthread_exit(NULL);
}
```

sharing.c

```c
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]:  %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}
```

Peer threads reference main thread's stack indirectly through global ptr variable

What can go wrong in concurrent programming

We will focus only on concurrency issues with multi-threading.

Common issues

- Data races / race conditions
- Deadlocks

Other issues that we omit in this lecture

- Starvation
- Contention / congestion
- Busy-wait loops (extensive resource usage)
- Priority-inversion

Example: What can go wrong (1)

We have a global variable:

```
int* a = malloc(sizeof(int));
*a = 5;
```

Now, two separate control flows (threads T1 and T2) use this variables concurrently.

T1:

```
int temp = *a;
temp += 3;
*a = temp;
```

T2:

```
int temp = *a;
temp += 4;
*a = temp;
```

We will call like this:

```
create_new_thread(T1);
create_new_thread(T2);
```

Will this work?

Example: What can go wrong (2)

```
// int* a = malloc(sizeof(int));
// *a = 5;
```

In reality, this may occur:

```
int temp = *a;          (T1)
temp += 3;              (T1)
int temp = *a;          (T2)
*a = temp;              (T1)
temp += 4;              (T2)
*a = temp;              (T2)
```

What is the result value of *a after both threads have run?

How can we fix this?

Example: What can go wrong (3)

– FIX: Introducing a mutex:

T1:

```
lock();
int temp = *a;
temp += 3;
*a = temp;
unlock();
```

T2:

```
lock();
int temp = *a;
temp += 4;
*a = temp;
```

Will this protect us?

Example: What can go wrong (4)

Oops.. we forget to unlock the mutex in T2. Imagine this control flow:

```
// int* a = malloc(sizeof(int));
// *a = 5;
```

```
lock();                (T2)
int temp = *a;         (T2)
temp += 4;             (T2)
*a = temp;             (T2)
lock();                (T1)
int temp = *a;         (T1)
temp += 3;             (T1)
*a = temp;             (T1)
unlock();              (T1)
```

What will happen?

Example: What can go wrong (5)

More complicated example (a common pitfall !) :

```
void modify()
{
     lock();
     int temp = *a;
     if(temp >= 8) return;
     temp += 3;
     *a = temp;
     unlock();
}
```

T1:

`modify();`

T2:

`modify();`

Is this correct?

Example: What can go wrong (6)

The problem:

```
void modify()
{
      lock();
      int temp = *a;
      if(temp >= 8) return;
      temp += 3;
      *a = temp;
      unlock();
}
```

The solution:

```
void modify()
{
      lock();
      int temp = *a;
      if(temp >= 8) {
            unlock();
            return;
      }
      temp += 3;
      *a = temp;
      unlock();
}
```

Condition variables? See video-recap on A5 on Absalon :)

Concurrency protection mechanisms

Fortunately, we have some powerful tools at our disposal to help us overcome these issues, provided we use them correctly:

Mutex

- Lock critical sections (Den her by er ik stor nok til os begge to)
- `pthread_mutex_(init|destroy|lock|unlock)`

Spinlock (extracurricular!)

- Fancy, locked busy-wait loop.

Condition variable

- Threads go to sleep waiting for a specific condition
- `pthread_cond_(init|destroy|wait|broadcast)`

Semaphore

- Like mutex, except N threads can lock simultaneously
- `sem_init`, `sem_destroy`, `sem_wait`, `sem_post`

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.