CMPSC 311 - Introduction to Systems Programming
Module: Concurrency

Professor Patrick McDaniel
Fall 2013
Sequential Programming

• Processing a network connection as it arrives and fulfilling the exchange completely is sequential processing
  ‣ i.e., connections are processed in *sequence* of arrival
Whither sequential?

• Benefits
  ‣ super simple to build
  ‣ very little persistent state to maintain

• Disadvantages
  ‣ incredibly poorly performing
    • one slow client causes *all* others to block
    • poor utilization of network, CPU
    • cannot interleave processing
Whither sequential?

• Benefits
  ‣ super simple to build
  ‣ very little persistent state to maintain

• Disadvantages
  ‣ incredibly poorly performing
    • one slow client causes all others to block
    • poor utilization of network, CPU
    • cannot interleave processing

Think about this way: if the class took the final exam sequentially, it would take 6 days and 16 hours to complete the exam!
An alternate design ...

- Why not process multiple requests at the same time, interleaving processing while waiting for other dependent actions (DB dips) to complete?
- This is known as *concurrent processing* ...
  - Process multiple requests concurrently

- We have/will explore three concurrency approaches
  1. Event programming using `select()` [see last lecture]
  2. Creating “child” processes using `fork()`
  3. Creating multithreaded programs using `pthreads`
Concurrency with processes

• The server process blocks on `accept()` , waiting for a new client to connect
  ‣ when a new connection arrives, the parent calls `fork()` to create another process
  ‣ the child process handles that new connection, and `exit()`’s when the connection terminates

• Children become “zombies” after death
  ‣ `Wait()` to “reap” children
Graphically

client

server
Graphically
Graphically

- Client
- Server
- Child

fork()
Graphically

```
client

server

server

fork() grandchild
```

CMPSC 311 - Introduction to Systems Programming

Page 10
Graphically

client → server

child exit( )ʼs / parent wait( )ʼs

server → server
Graphically

client → server

server

parent closes its client connection
Graphically
Graphically
Graphically
Graphically
fork()

- The `fork` function creates a new process by duplicating the calling process.
  ```c
  pid_t fork(void);
  ```
- The new **child** process is an exact duplicate of the calling **parent** process, except that it has its own process ID and pending signal queue.
- The `fork()` function returns
  - 0 (zero) for the child process OR ...
  - The child’s `process ID` in the parent code.

Idea: think about duplicating the process state and running ....
Process control

- **Parent**
  - fork (pid == child PID)
  - wait for child to complete (maybe)

- **Child**
  - begins at fork (pid == 0)
  - runs until done
  - calls exit
exit()

- The `exit` causes normal process termination with a return value
  
  ```c
  void exit(int status);
  ```

- Where
  - `status` is sent to the parent (\&0377)

- Note: `exit` vs. `return` from a C program
  - `return` simply returns a normal execution
  - `exit` kills the process via system call
  - Consequence: `return` is often cleaner for certain things that run `atexit` (more of a C++ concern)
wait()

- The `wait` function is used to wait for state changes in a child of the calling process (e.g., to terminate)
  
  ```c
  pid_t wait(int *status);
  ```

- Where
  - returns the process ID of the child process
  - `status` is return value set by the child process
wait() and the watchdog

- Often you will want to safeguard a process by “watching” it execute. If the process terminates then you take some action (when `wait()` returns).
  - This is called a **watchdog process** (or watchdog fork)
  - Idea: you fork the process and simply wait for it to terminate
    - Take some action based on the idea that the termination should not have happened or that some recovery is needed.
      - e.g., restart a web server when it crashes

- Watchdogs are very often used for critical services.
Putting it all together ...

```c
int main( void ) {

    pid_t childpid; // Child process ID
    int status;     // Childs exist status

    printf( "Stating our process control example ...
"
             );
    childpid = fork(); // Fork the process

    if ( childpid >= 0 ) {
        if ( childpid == 0 ) { // In child process
            printf( "Child processing,\n"
                    );
            sleep( 1 );
            exit( 19 );
        } else { // In parent process
            printf( "Parent processing (child=%d),\n", childpid );
            wait( &status );
            printf( "Child exited with status=%d.\n", WEXITSTATUS(status) );
        }
    } else { // in calling process
        perror( "What the fork happened" );
        exit( -1 );
    }

    return( 0 );
}
```
int main( void ) {

    pid_t childpid;  // Child process ID
    int status;      // Childs exist status

    printf( "Stating our process control example ...\n" );
    childpid = fork();  // Fork the process

    if ( childpid >= 0 ) {
        if ( childpid == 0 ) {  // In child process
            printf( "Child processing,\n" );
            sleep( 1 );
            exit( 19 );
        } else {  // In parent process
            printf( "Parent processing (child=%d),\n", childpid );
            wait( &status );
            printf( "Child exited with status=%d.\n", WEXITSTATUS(status) );
        }
    } else {
        perror( "What the fork happened" );
        exit( -1 );
    }

    return( 0 );
}

$ ./forker
Stating our process control example ...
Parent processing (child=3530),
Child processing.
Child exited with status=19.
$
Process Control Tradeoffs

• Benefits
  ‣ almost as simple as sequential
  ‣ in fact, most of the code is identical!
  ‣ parallel execution; good CPU, network utilization
  ‣ often better security (isolation)

• Disadvantages
  ‣ processes are heavyweight
  ‣ relatively slow to fork
  ‣ context switching latency is high
  ‣ communication between processes is complicated
Concurrency with threads

• A single process handles all of the connections
  ‣ but, a *parent thread* forks (or dispatches) a new thread to handle each connection
  ‣ the *child thread*:
    • handles the new connection
    • exits when the connection terminates

*Note*: you can create as many threads as you want (up to a system limit)
Threads

• A thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system.
  ‣ To the software developer, the concept of a "procedure" that runs independently from its main program.
  ‣ To go one step further, imagine a main program that contains a number of procedures. Now imagine all of these procedures being able to be scheduled to run simultaneously and/or independently by the operating system. That would describe a "multi-threaded" program.

Idea: “forking” multiple threads of execution in one process!
Threads (cont.)

server

accept( )
Threads (cont.)

client

connect

accept()

server
Threads (cont.)

client

pthread_create()

server
Threads (cont.)
Threads (cont.)

client

client

pthread_create()

server
Threads (cont.)

- Clients communicate with the server, which has shared data structures.

Diagram:

- Clients
- Server
- Shared data structures
- Arrows indicate communication directions.
Threads (cont.)

UNIX Process

... and with threads
Threads (cont.)

• This independent flow of control is accomplished because a thread maintains its own:
  ‣ Stack pointer
  ‣ Registers
  ‣ Scheduling properties (such as policy or priority)
  ‣ Set of pending and blocked signals
  ‣ Thread specific data.
Thread Summary

• Exists within a process and uses the process resources
• Has its own independent flow of control as long as its parent process exists and the OS supports it
•Duplicates only the essential resources it needs to be independently “schedulable“
• May share the process resources with other threads that act equally independently (and dependently)
• Dies if the parent process dies - or something similar
• Is "lightweight" because most of the overhead has already been accomplished through the creation of its process.
Caveots

• Because threads within the same process share resources:
  ‣ Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads.
  ‣ Two pointers having the same value point to the same data.
  ‣ Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer.
Thread control

• main
  ‣ `pthread_create()` (create thread)
  ‣ `wait for thread()` to finish via `pthread_join` (maybe)

• thread
  ‣ begins at function pointer
  ‣ runs until `pthread_exit()`
The pthread_create function starts a new thread in the calling process.

```c
int pthread_create(pthread_t *thread,
                  const pthread_attr_t *attr,
                  void *(*start_routine) (void *),
                  void *arg);
```

Where,

- **thread** is a pthread library structure holding thread info
- **attr** is a set of attributes to apply to the thread
- **start_routine** is the thread function pointer
- **arg** is an opaque data pointer to pass to thread
**pthread_join()**

- The `pthread_join` function waits for the thread specified by `thread` to terminate.

```c
int pthread_join(pthread_t thread, void **retval);
```

- Where,
  - `thread` is a pthread library structure holding thread info
  - `retval` is a double pointer return value
pthread_exit()

- The pthread_exit function terminates the calling thread and returns a value

  ```c
  void pthread_exit(void *retval);
  ```

- Where,
  - `retval` is a pointer to a return value
    - **Note:** better be dynamically allocated because the thread stack will go away when the thread exits
typedef struct {
    int num;
    const char *str;
} MY_STRUCT;

void * thread_function( void * arg ) {
    MY_STRUCT *val = (MY_STRUCT *)arg; // Cast to expected type
    printf( "Thread %lx has values %x,%s\n", pthread_self(), val->num, val->str );
    pthread_exit( &val->num );
}

int main( void ) {
    MY_STRUCT v1 = { 0x12345, "Val 1" };
    MY_STRUCT v2 = { 0x54312, "Val 2" };
    pthread_t t1, t2;
    printf( "Starting threads\n" );
    pthread_create( &t1, NULL, thread_function, (void *)&v1 );
    pthread_create( &t2, NULL, thread_function, (void *)&v2 );
    pthread_join( t1, NULL );
    pthread_join( t2, NULL );
    printf( "All threads returned\n" );
    return( 0 );
}
### Putting it all together ...

```c
typedef struct {
    int num;
    const char *str;
} MY_STRUCT;

void * thread_function( void * arg ) {
    MY_STRUCT *val = (MY_STRUCT *)arg; // Cast to expected type
    printf( "Thread %lx has vaules %x,%s\n", pthread_self(), val->num, val->str );
    pthread_exit( &val->num );
}

int main( void ) {
    MY_STRUCT v1 = { 0x12345, "Val 1" };
    MY_STRUCT v2 = { 0x54312, "Va1 2" };
    pthread_t t1, t2;
    printf( "Starting threads\n" );
    pthread_create( &t1, NULL, thread_function, (void *)&v1 );
    pthread_create( &t2, NULL, thread_function, (void *)&v2 );
    pthread_join( t1, NULL );
    pthread_join( t2, NULL );
    printf( "All threads returned\n" );
    return( 0 );
}
```

$ ./concurrency
Starting threads
Thread 7f51c3e05700 has vaules 54312,Val 2]
Thread 7f51c4606700 has vaules 12345,Val 1]
All threads returned
$
```
Thread pooling

- A systems design in which a number of threads are pre-created and assigned to requests/tasks as needed
  - Once they are done processing, they go back on the queue

**Idea:** amortize the cost of thread creation over many requests
Challenge ...

- Consider the following function:

- Now suppose thread “a” executes the following

- ... at the same time thread “b” executes

- What is the final value of accounts[1]?
Challenge ...

• Consider the following function:

```c
int accounts[5] = { 0, 0, 0, 0, 0 };
int addvalue( int account, int amount ) {
    int num = accounts[account];
    num = num+amount;
    accounts[account] = num;
    return( 0 );
}
```

• Now suppose thread “a” executes the following

• ... at the same time thread “b” executes

• What is the final value of `accounts[1]`?
Challenge ... 

• Consider the following function:

```c
int accounts[5] = { 0, 0, 0, 0, 0 };  
int addvalue( int account, int amount ) {
    int num = accounts[account];  
    num = num+amount;  
    accounts[account] = num;  
    return( 0 );
}
```

• Now suppose thread “a” executes the following:

```c
addValue( 1, 100 );
```

• ... at the same time thread “b” executes

• What is the final value of `accounts[1]`?
Challenge ...

• Consider the following function:

```c
int accounts[5] = { 0, 0, 0, 0, 0 };  
int addvalue( int account, int amount ) {
    int num = accounts[account];
    num = num + amount;
    accounts[account] = num;
    return( 0 );
}
```

• Now suppose thread “a” executes the following

```c
addValue( 1, 100 );
```

• ... at the same time thread “b” executes

```c
addValue( 1, 200 );
```

• What is the final value of accounts[1]?
Watch the execution!

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
<th>num &quot;A&quot;</th>
<th>num &quot;B&quot;</th>
<th>accounts[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>int num = accounts[account];</td>
<td>int num = accounts[account];</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>int num = accounts[account];</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>num = num+amount;</td>
<td></td>
<td></td>
<td>200</td>
<td></td>
</tr>
<tr>
<td>accounts[account] = num;</td>
<td>accounts[account] = num;</td>
<td></td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>num = num+amount;</td>
<td></td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>accounts[account] = num;</td>
<td></td>
<td>100</td>
<td></td>
<td>100</td>
</tr>
</tbody>
</table>

- **Q:** What happened?
- **A:** The temporary value of shared held stomped on the other thread’s data/computation.
  - This is known as a race condition.
- The OS course will teach you how to handle this using synchronization via MUTEXes (mutual exclusion).
Thread tradeoffs

• Benefits
  ‣ straight-line code, line processes or sequential
  ‣ still the case that much of the code is identical!
  ‣ parallel execution; good CPU, network utilization
  ‣ lower overhead than processes
  ‣ shared-memory communication is possible

• Disadvantages
  ‣ synchronization is complicated
  ‣ shared fate within a process; one rogue thread can hurt you
  ‣ security (no isolation)