POSIX / System Programming

Guest lecture – 2017-10-06

Carlos Moreno
cmoreno@uwaterloo.ca
E5-4111
Outline

• During today's lecture, we'll look at:

  • Some of POSIX facilities
    • Main focus on processes, concurrency, communication, threads and synchronization.
    • Issues with concurrency: race conditions, deadlock, starvation.
    • Tools and techniques to deal with the above: critical sections, mutual exclusion / atomicity, semaphores, pipes, message queues, shared memory.
One of the most important notions is that of a *Process*.

Possible definitions:

- A program in execution / An instance of a program running on a computer
  - Not really: execution of a program can involve multiple processes!
- A unit of activity characterized by the execution of a sequence of instructions, a current state, and an associated set of system instructions
Process

- An entity representing activity consisting of three components:
  - An executable program
  - Associated data needed by the program
  - Execution context of the program (registers, PC, pending I/O operations, etc.)
- OS assigns a unique identifier (PID)
  - See command `ps`.
- Processes can create other processes (denoted "child process" in that context)
  - See `ps --forest`
Multiprogramming

- Concurrent execution of multiple tasks (e.g., processes)
  - Each task runs as if it was the only task running on the CPU.

- Benefits:
  - When one task needs to wait for I/O, the processor can switch to the another task.
  - (why is this potentially a huge benefit?)
Multiprogramming

(c) Multiprogramming with three programs
Multiprogramming

• Example / case-study:
  • Demo of web-based app posting jobs and a simple command-line program processing them.
    • Can run multiple instances of the job processing program.
    • Or we can have the program use `fork()` to spawn multiple processes that work concurrently.
Multithreading

- Processes typically have their own “isolated” memory space.
  - Memory protection schemes prevent a process from accessing memory of another process (more in general, any memory outside its own space).
  - The catch: if processes need to share data, then there may be some overhead associated with it.

- Threads are a “lighter version” of processes:
  - A process can have multiple threads of execution that all share the same memory space.
  - Sharing data between threads has little or no overhead
    - Good news? Bad news? (both?)
Multithreading

• Example/demo:
  • With the multithreading demo, we'll look at a different application/motivation for the use of concurrency: performance boost through parallelism.
    • Possible when we have multiple CPUs (e.g., multicore processors)
    • Important to have multiple CPUs when the application is CPU-bound.
Concurrency Issues

• Race condition:

A situation where concurrent operations access data in a way that the outcome depends on the order (the timing) in which operations execute.

• Doesn't necessarily mean a bug! (like in the threads example with the linked list)

• In general it constitutes a bug when the programmer makes any assumptions (explicit or otherwise) about an order of execution or relative timing between operations in the various threads.
Concurreny Issues

- Race condition:

Example (x is a shared variable):

Thread 1: Thread 2:

x = x + 1; x = x – 1;

(what's the implicit assumption a programmer could make?)
Concurrency Issues

- **Race condition:**
  
  Thread 1:
  
  \[ x = x + 1; \]

  Thread 2:
  
  \[ x = x - 1; \]

- **In assembly code:**
  
  R1 ← x  
  inc R1  
  R1 → x  

  R1 ← x  
  dec R1  
  R1 → x
Concurrency Issues

- And this is how it could go wrong:
  
  Thread 1:
  \[ x = x + 1; \]
  
  Thread 2:
  \[ x = x - 1; \]

- In assembly code:
  
  \[
  \begin{align*}
  R1 & \leftarrow x \\
  \text{inc} & \ R1 \\
  R1 & \rightarrow x \\
  \end{align*}
  \]

  \[
  \begin{align*}
  R1 & \leftarrow x \\
  \text{dec} & \ R1 \\
  R1 & \rightarrow x \\
  \end{align*}
  \]
Concurrency Issues

• Atomicity / Atomic operation:

  Atomicity is a characteristic of a fragment of a program that exhibits an observable behaviour that is non-interruptible – it behaves as if it can only execute entirely or not execute at all, such that no other threads deal with any intermediate outcome of the atomic operation.

  • Non-interruptible applies in the context of other threads that deal with the outcome of the operation, or with which there are race conditions.

  • For example: in the pthreads demo, if the insertion of an element in the list was atomic, there would be no problem.
Concurreny Issues

• Examples of atomic operations in POSIX:
  
  • Renaming / moving a file with
    `int rename (const char * old, const char * new);`
    Any other process can either see the old file, or
    the new file – not both and no other possible
    “intermediate” state.

  • `open`ing a file with attributes `O_CREAT` and `O_EXCL`
    (that is, creating a file with exclusive access). The
    operation atomically attempts to create the file: if it
    already exists, then the call returns a failure code.
Concurrency Issues

• Mutual Exclusion:

  Atomicity is often achieved through mutual exclusion – the constraint that execution of one thread excludes all the others.

  • In general, mutual exclusion is a constraint that is applied to sections of the code.

  • For example: in the pthreads demo, the fragment of code that inserts the element to the list should exhibit mutual exclusion: if one thread is inserting an element, no other thread should be allowed to access the list

  • That includes main, though not a problem in this particular case (why?)
Concurrency Issues

• Critical section:

A section of code that requires atomicity and that needs to be protected by some mutual exclusion mechanism is referred to as a critical section.

• In general, we say that a program (a thread) enters a critical section.
Concurrent Issues

• Mutual Exclusion – How?

Attempt #1: We disable interrupts while in a critical section (and of course avoid any calls to the OS)

• There are three problems with this approach
  • Not necessarily feasible (privileged operations)
  • Extremely inefficient (you're blocking everything else, including things that wouldn't interfere with what your critical section needs to do)
  • *Doesn't always work!!* (keyword: multicore)
Concurreny Issues

• Mutual Exclusion – How?

Attempt #2: We place a flag (sort of telling others “don't touch this, I'm in the middle of working with it).

    int locked; // shared between threads
    // ...
    if (! locked)
    {
        locked = 1;
        // insert to the list (critical section)
        locked = 0;
    }

• Why is this flawed? (there are several issues)
Concurrency Issues

• Mutual Exclusion – How?

One of the problems: does not really work!

This is what the assembly code could look like:

R1 ← locked
tst R1
brnz somewhere_else
R1 ← 1
R1 → locked
Concurrent Issues

- Mutual Exclusion – How?

Another problem: an if statement just doesn't cut it! We need to insert an element – if some other thread is inserting an element at this time, we need to wait until the other thread finishes:

```java
while (locked) {}
locked = 1;
// ... critical section
locked = 0;
```

There are two problems with this: one is that it doesn't work (for the same reason as with the if). What's the other problem?
Concurrency Issues

• Mutex:

A mutex (for MUTual EXclusion) provides a clean solution: In general we have a variable of type mutex, and a program (a thread) attempts to lock the mutex. The attempt atomically either succeeds (if the mutex is unlocked) or it blocks the thread that attempted the lock (if the mutex is already unlocked).

• As soon as the thread that is holding the lock unlocks the mutex, this thread's state becomes ready.
Concurrent Issues

• Using a Mutex:

lock (mutex)

\textit{critical section}

unlock (mutex)

• For example, with POSIX threads (pthreads):

\begin{verbatim}
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
// ...
pthread_mutex_lock (&mutex);
// ... \textit{critical section}
pthread_mutex_unlock (&mutex);
\end{verbatim}
Concurrency Issues

• Using a Mutex:
  • One issue is that POSIX only defines mutex facilities for threads --- not for processes!
  • We could still implement it through a “lock file” (created with `open` using flags `O_CREAT` and `O_EXCL`)
    • Not a good solution (it *does* work, but is has the same issues as the lock variable example)
Concurrency and synchronization

- Another synchronization primitive: Semaphores

(image courtesy of wikipedia.org)
Concurrency and synchronization

• Another synchronization primitive: Semaphores

  • Semaphore: A counter with the following properties:
    • Atomic operations that increment and decrement the count
    • Count is initialized with a non-negative value
    • `wait` operation decrements count and causes caller to block if count becomes negative (if it was 0)
    • `signal` (or `post`) operation increments count. If there are threads blocked (waiting) on this semaphore, it unblocks one of them.
Concurrent Programming and Synchronization

- Producer / consumer with semaphores

```c
semaphore items = 0;
muxet_t mutex; // why also a mutex?

void producer()
{
    while (true)
    {
        produce_item();
        lock (mutex);
        add_item();
        unlock (mutex);
        sem_signal (items);
    }
}

void consumer()
{
    while (true)
    {
        sem_wait (items);
        lock (mutex);
        retrieve_item();
        unlock (mutex);
        consume_item();
    }
}
```
Concurrency and synchronization

- Mutual Exclusion with semaphores
  - Interestingly enough – Mutexes can be implemented in terms of semaphores!

```c
semaphore lock = 1;

void process ( ... )
{
    while (1)
    {
        /* some processing */
        sem_wait (lock);
        /* critical section */
        sem_signal (lock);
        /* additional processing */
    }
}
```
Concurrency and synchronization

- Producer / consumer with semaphores only

```c
semaphore items = 0;
semaphore lock = 1;

void producer()
{
    while (true)
    {
        produce_item();
        sem_wait (lock);
        add_item();
        sem_signal (lock);
        sem_signal (items);
    }
}

void consumer()
{
    while (true)
    {
        sem_wait (items);
        sem_wait (lock);
        retrieve_item();
        sem_signal (lock);
        consume_item();
    }
}
```
Concurrency and synchronization

- Producer / consumer with semaphores only
  - Interestingly, POSIX does provide inter-process semaphores!
Concurrency and synchronization

- POSIX semaphores:
  - Defined through data type sem_t
  - Two types:
    - Memory-based or unnamed (good for threads)
    - Named semaphores (system-wide — good for processes synchronization)
Concurrency and synchronization

- POSIX semaphores:
  - For unnamed semaphores:
    - Declare a (shared – possibly as global variable) `sem_t` variable
    - Give it an initial value with `sem_init`
    - Call `sem_wait` and `sem_post` as needed.

```c
sem_t items;
sem_init (&items, 0, initial_value);
// ...
sem_wait (&items) or sem_post (&items)
```
Concurrent access to shared resources requires synchronization.

- **POSIX semaphores:**
  - For named semaphores:
    - Similar to dealing with a file: have to “open” the semaphore – if it does not exist, create it and give it an initial value.

```c
sem_t * items = sem_open (semaphore_name, flags, permissions, initial_value);
// should check if items == SEM_FAILED

// ...

sem_wait (items) or sem_post (items)
```
Concurrency and synchronization

• Producer-consumer:
  • We'll work on the example of the web-based demo as a producer-consumer with semaphores.
  • Granularity for locking?
    • Should we make the entire process_requests a critical section?
      • Clearly overkill! No problem with two separate processes working each on a different file!
      • We can lock the file instead — no need for a mutex, since this is a consumable resource.
      • For a reusable resource, we'd want a mutex – block while being used, but then want to use it ourselves!
Concurrency and synchronization

• Consumable vs. Reusable Resource:
  • With a consumable resource, we want to:
    • Try to lock it.
      • If failed, then forget about it (someone else locked it and will make it disappear – will “consume” it).
  • With a reusable resource:
    • Wait until you can lock it (as in, attempt to lock it blocking if it is already locked)
      • When unlocked by the other thread/process, then we lock it and (re)use it.
Concurrenty and synchronization

- **Consumable vs. Reusable Resource:**
  - See code for demo — locking has to be done atomically!
  - We recall that renaming a file with `rename` is an atomic operation!
Concurrency and synchronization

• More on locking granularity:

• Consider the following scenario:
  • One thread writes to some shared resource (e.g., a linked list)
  • Many threads need to read that shared resource
    • Observation: concurrent reads don't cause a race condition (right?)
    • Do we need to lock the resource when reading?
Concurrency and synchronization

- More on locking granularity:
- Consider the following scenario:
  - Problem:
    - Concurrent reads do not need mutual exclusion
    - But since a write could be taking place, we need to define the read operation as a critical section, in case there is a concurrent write operation!
Concurrency and synchronization

• More on locking granularity:

• Consider the following scenario:
  
  • Problem:
    • Concurrent reads do not need mutual exclusion
    • But since a write could be taking place, we need to define the read operation as a critical section, in case there is a concurrent write operation!

  • Solution:
    • Finer granularity!
      • Locking for write vs. locking for read!
Concurrency and synchronization

- More on locking granularity:
  - Read/Write locks implement this functionality:
    - Threads calling read_lock do not exclude each other.
    - A thread calling write_lock excludes any other threads requesting write_lock and also any other threads requesting read_lock
      - It blocks if some thread is holding a read lock!
Concurrence and synchronization

• More on locking granularity:

• Read/Write locks implement this functionality:
  • Threads calling read_lock do not exclude each other.
  • A thread calling write_lock excludes any other threads requesting write_lock and also any other threads requesting read_lock
    • It blocks if some thread is holding a read lock!

• POSIX R/W Locks:

  pthread_rwlock_t
  pthread_rwlock_rdlock ( ... )
  pthread_wrlock_wrlock ( ... )
  pthread_rwlock_unlock ( ... )
Concurrency and synchronization

- More on locking granularity:
- Big problem with Read/Write locks?
  - Hint: what happens if many threads are reading very frequently?
Concurrency and synchronization

- **Starvation:**
  - One of the important problems we deal with when using concurrency:
  - An otherwise ready process or thread is deprived of the CPU (it's starved) by other threads due to, for example, the algorithm used for locking resources.
    - Notice that the writer starving is not due to a defective scheduler/dispatcher!
Concurrenty – Deadlock

• Deadlock:
  • Consider the following scenario:
  • A Bank transaction where we transfer money from account A to account B
  • Clearly, there is a (dangerous) race condition
    • Want granularity — can not lock the entire bank so that only one transfer can happen at a time
    • We want to lock at the account level:
      • Lock account A, lock account B, then proceed!
Concurrency – Deadlock

- Deadlock:
  - Problem with this?
  - Two concurrent transfers — one from account 100 to account 200, one from account 200 to account 100.
    - If the programming is written as:
      Lock source account
      Lock destination account
      Transfer money
      Unlock both accounts
Concurrency – Deadlock

• Deadlock:
  • Problem with this?
  • Two concurrent transfers — one from account 100 to account 200, one from account 200 to account 100.
    • Process 1 locks account 100, then locks account 200
    • Process 2 locks account 200, then locks account 100
Concurrency – Deadlock

- Deadlock:
  - What about the following interleaving?
    - Process 1 locks account 100
    - Process 2 locks account 200
    - Process 1 attempts to lock account 200 (blocks)
    - Process 2 attempts to lock account 100 (blocks)
  - When do these processes unblock?
Concurrency – Deadlock

• Deadlock:
  • What about the following interleaving?
    • Process 1 locks account 100
    • Process 2 locks account 200
    • Process 1 attempts to lock account 200 (blocks)
    • Process 2 attempts to lock account 100 (blocks)

• When do these processes unblock?

• Answer: under some reasonable assumptions, never!
Concurrency – Deadlock

• Deadlock:
  • Graphically:
Concurrency – Deadlock

• Deadlock:

  - Solution in this case is really simple:
    - Lock the resources in a given order (e.g., by ascending account number).
Interprocess Communication

• Sharing data between processes:
  • Requires synchronization (to avoid race conditions, and to access data when there is data to be accessed!)
  • Typical mechanisms:
    • Through designated files (obvious, but inefficient)
    • Through pipes (very simple, but limited)
    • Through shared memory (efficient, but dangerous!)
    • Through message queues (convenient, though not particularly simple)
Interprocess Communication

• Sharing data through files:
  • Not much to say – one process writes data to a file, another process reads data from the file.
  • Still need synchronization
Interprocess Communication

- **Pipes:**
  - A pipe is a mechanism to set up a “conduit” for data from one process to another.
  - It is unidirectional (i.e., we have to predefine who transmits and who receives data)
  - Simplest form is with `popen`:
    - It executes a given command (created as a child process) and returns a stream (a FILE *) to the calling process:
    - It then connects either the standard output of that command to the (input) stream, or the standard input of that command to the (output) stream.
Interprocess Communication

• Pipes – example:

  • To read the output from a program:

    ```c
    FILE * child = popen ("/path/command", "r");
    if (child == NULL)  { /* handle error condition */ } 
    
    Now read data with, e.g., fread ( ... , ... , ... , child);
    and NEVER forget to pclose (child);
    ```

  • Whatever data the child process sends to its standard output (e.g., with printf) will be read by the parent.

  • Conversely, if we popen ( .... , “w”), then whatever data we write to it (e.g., with fprintf or fwrite) will appear through the standard input of the child.
Interprocess Communication

• Pipes:
  • For more details, see `man popen`
  • For the more general form, including named pipes, see `man 7 pipe` and `man 2 pipe`. 
Interprocess Communication

- **Shared memory:**
  
  - Mechanism to create a segment of memory and give multiple processes access to it.

  - `shmget` creates the segment and returns a handle to it (just an integer value)

  - `shmat` creates a logical address that maps to the beginning of the segment so that this process can use that memory area

    - If we call `fork()`, the shared memory segment is inherited shared (unlike the rest of the memory, for which the child gets an independent copy)
Interprocess Communication

- Shared memory:
  - For more information, see `man shmget` and `man shmat`
Interprocess Communication

• **Message queues:**
  
  • Mechanism to create a queue or “mailbox” where processes can send messages to or read messages from.
  
  • `mq_open` opens (creating if necessary) a message queue with the specified name.
  
  • `mq_send` and `mq_receive` are used to transmit or receive (receive by default blocks if the queue is empty) from the specified message queue.
Interprocess Communication

- **Message queues:**
  - Big advantages:
    - Allows multiple processes to communicate with other multiple processes
    - Synchronization is somewhat implicit!
  - See `man mq_overview` for details.