CSC2/452 Computer Organization
Concurrency and Inter-process Communication

Sreepathi Pai

URCS

November 13, 2019

Outline

Outline

Administrivia

- Homework #6 is out
  - Some printouts still with me
  - Due Monday, Nov 18, in class
- Assignment #4 is out
  - Due date: Tuesday, Nov 26, 7PM
- Assignment #5 (last!) will be out Dec 2

Outline

Splendid Isolation?

- Processes are isolated from other processes
  - CPU + OS enforce this
- But if processes can't exchange data, work cannot be split
  - Would be unable to do work in parallel
- Unfettered sharing is also dangerous
  - Lots of security problems
- How do we share data in a controlled manner?

Outline

The problems of data sharing

Data sharing requires:

- A shared medium
  - obvious requirement
- A mechanism for synchronization
  - i.e. ordering or mutual exclusion
  - this is used to order/control accesses to the shared medium

Shared medium

What is shared between processes?

- Pipeline?
- Cache/Memory Hierarchy?
- Disk/Filesystem?

Shared medium

- Pipeline is time-shared, and CPUs isolate the pipeline from different processes
  - apparently, not very successfully, as recent revelations show
  - ZombieLoad
- Cache/memory hierarchy is space-shared, but uses virtual addressing to isolate processes
  - Processes don't share address space by default, so can't locate each others data
  - again, side-channels (usually timing), can be used to leak data
- Disk/filesystem?
  - Shared address space (filename)

Sharing data through the filesystem

- Process A writes data to file
- Process B reads data from file
- Process A and Process B can be running at the same time

Process Ordering

- Recall that concurrently running processes get a slice of the CPU
  - Usually 100ms
- The OS decides the order in which processes are executed by the CPU
  - This order is *non-deterministic*

Example

```
for(int i = 0; i < nchild; i++) {
  if(fork() == 0) {
    printf("In child %d\n", i);
    return 0;
  }
}
```

Output for 3

```
$ ./fork_order 3
Creating 3 child processes
In child 0
In child 1
In child 2
```

Output for 7

```
$ ./fork_order 7
Creating 7 child processes
In child 0
In child 1
In child 3
In child 2
In child 4
In child 6
In child 5
```

Adding N numbers

```
/* a is an array of N elements */
NPERCHILD = (N+nchild-1)/nchild;

unsigned int sum = 0;

for(int i = 0; i < nchild; i++) {
  if(fork() == 0) {
    printf("In child %d, adding array elements from %d\n", i,
       i * NPERCHILD);

    for(int j = i * NPERCHILD;
        j < (i * NPERCHILD + NPERCHILD) && j < N;
        j++)
    {
      sum += a[j];
    }

    printf("In child %d, sum is %d\n", i, sum);
    return 0;
  }
}

printf("In parent, sum is %d\n", sum);
```

- Each child process computes the sum of part of an array

Output

```
Creating 5 child processes to add 10000 numbers
In child 0, adding array elements from 0
In child 0, sum is 1999000
In child 1, adding array elements from 2000
In child 1, sum is 5999000
In parent, sum is 0
In child 2, adding array elements from 4000
In child 3, adding array elements from 6000
In child 4, adding array elements from 8000
In child 4, sum is 17999000
In child 2, sum is 9999000
In child 3, sum is 13999000
```

- Problem #1: Parent is not ordered with respect to child processes

Ordering Parent w.r.t. Child Processes

```
/* loop that forks child processes */

int pid = 0;
int wstatus = 0;
while(1) {
  pid = waitpid(-1, &wstatus, 0);
  if(pid == -1) {
    if(errno == ECHILD) break; // no more child processes left
    if(errno == EINTR) continue;
  }
}
printf("In parent, sum is %d\n", sum);
```

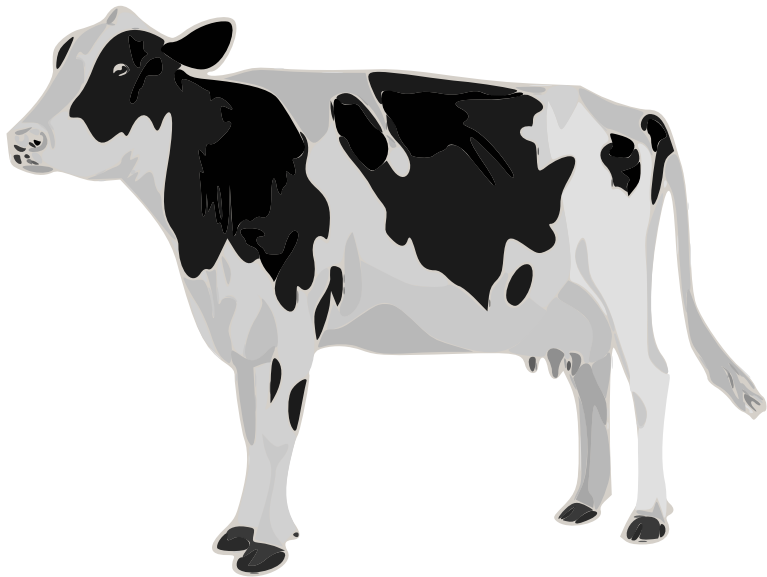
- The loop waits for all child processes

Output after ordering

```
Creating 5 child processes to add 10000 numbers
In child 0, adding array elements from 0
In child 0, sum is 1999000
In child 2, adding array elements from 4000
In child 1, adding array elements from 2000
In child 1, sum is 5999000
In child 2, sum is 9999000
In child 4, adding array elements from 8000
In child 4, sum is 17999000
In child 3, adding array elements from 6000
In child 3, sum is 13999000
In parent, sum is 0
```

- This is ordered, but sum is still 0
  - It should be 49995000
- Why?

C-O-W



- Although fork() duplicates data, it is copy-on-write
  - Any writes will not be shared!

Adding shared memory

```
unsigned int *sum;
sum = mmap(NULL, 4096, PROT_READ | PROT_WRITE,
           MAP_SHARED | MAP_ANONYMOUS, -1, 0);

if(sum == MAP_FAILED) {
  perror("mmap");
  exit(1);
}

*sum = 0; // not required, since mmap initializes to 0

/* fork loop follows */
```

- We use `MAP_SHARED | MAP_ANONYMOUS` to create a shared page, and store the pointer to that page in `sum`
- All child processes will share that page too, in read/write mode
  - `*sum += a[i]`

Output

```
Creating 5 child processes to add 10000 numbers
In child 0, adding array elements from 0
In child 0, sum is 1999000
In child 1, adding array elements from 2000
In child 1, sum is 7998000
In child 2, adding array elements from 4000
In child 3, adding array elements from 6000
In child 3, sum is 21997000
In child 4, adding array elements from 8000
In child 2, sum is 17997000
In child 4, sum is 35996000
In parent, sum is 35996000
```

- What happened?

Output, again

```
Creating 5 child processes to add 10000 numbers
In child 0, adding array elements from 0
In child 0, sum is 1999000
In child 2, adding array elements from 4000
In child 1, adding array elements from 2000
In child 1, sum is 7998000
In child 2, sum is 11998000
In child 4, adding array elements from 8000
In child 4, sum is 29997000
In child 3, adding array elements from 6000
In child 3, sum is 43996000
In parent, sum is 43996000
```

- The results are different!
  - Program is executing non-deterministically

Code

```
for(int j = i * NPERCHILD;
    j < (i * NPERCHILD + NPERCHILD) && j < N;
    j++)
{
  *sum += a[j];
}
```

- Addition is associative
  - Order shouldn't matter!

Dissecting that line

```
*sum += a[j]
```

- Read the contents of `a[j]` and add them to the value at address pointed to by `sum`
- What is our expectation about the execution of this statement?

Assembly language code

```
  mov    0x0(%r13),%eax        # eax = *sum

loop:
  movslq %r12d,%rcx            # rcx = j
  add    (%rbx,%rcx,4),%eax    # eax += rbx[rcx*4]

  ... check if loop is over
  jl loop                      # j < ...

  mov    %eax,0x0(%r13)        # *sum = eax
```

- C decided reading/writing to memory on every iteration of the loop was too slow
  - So it read `*sum` once at beginning of loop, and stored it in `%eax`
  - It is permitted to make copies like this for variables that are not shared
- As a result, what happens when different child processes write their values of `eax` to `sum`?

A more subtle issue

- It currently takes three instructions
  - one to load `*sum` into a register
  - one to add `a[j]` to the register
  - one to store `*sum` back into memory
- You could be interrupted between any of those instructions!
  - You might be operating on stale values

Solving these issues

- How to prevent the C compiler from storing values in registers?
  - I.e. how to make it always read/write from memory?
- How to execute the group of instructions atomically?
  - I.e. load and add and store should behave like *one* operation

C11 feature

```
#include <stdatomic.h>
...
    atomic_unsigned_int *sum;
```

- Change sum's type to `atomic_unsigned_int *`
- Compile with `gcc -std=gnu11` (or `gcc -std=c11`, but you won't get `MAP_ANONYMOUS`)

```
loop:
  movslq %r12d,%rdx              % rdx = j
  mov    (%rbx,%rdx,4),%edx      % edx = rbx[rdx*4]
  lock add %edx,0x0(%r13)        % *sum += edx
  ...
```

- `*sum` is no longer read into register
- `lock` prefix added to add instruction
  - Other processes can't access the cache line containing `*sum` when add is executing

Output

```
Creating 5 child processes to add 10000 numbers
In child 0, adding array elements from 0
In child 1, adding array elements from 2000
In child 0, sum is 1999000
In child 2, adding array elements from 4000
In child 1, sum is 7998000
In child 4, adding array elements from 8000
In child 2, sum is 20782970
In child 4, sum is 35996000
In child 3, adding array elements from 6000
In child 3, sum is 49995000
In parent, sum is 49995000
```

- It works!
  - Or does it?
  - Are we just seeing one order where the answer was correct?

C11 atomics

- The compiler needs to be told that some variables are *shared*
- No way to do this reliably before the C11 standard
  - Some hodgepodge of `volatile` and machine-specific assembly code
- C11 brings `atomic` variables to the C language
  - Imply that the variable is shared
  - Can recognize certain composite operations as "atomic" and generates appropriate assembly

Warning

The code, while parallel (and correct), is not necessarily fast.

- You should update atomic variables as few times as possible
  - Compute a private sum in a unshared variable (i.e. register), and then add it to the shared sum
- Parallel sum has a better algorithm
- But take CS2/458 to learn more about these issues

Outline

Problem

```
// transfer 10000 from account a to b

balance_a -= 10000
                      <---- interrupted here
balance_b += 10000
```

- Each operation here is atomic
- But, logically, the entire *transfer* should be atomic
- Assuming `balance_a`=15000 and `balance_b`=15000 before the transfer
  - a valid state after is `balance_a`=5000 and `balance_b`=25000
- An invalid (logical) state is `balance_a`=5000 and `balance_b`=15000
  - e.g. if the program was context-switched at the indicated point

Atomics for non-primitive types

- C11 atomics (and atomics in general) work only on primitive types
  - i.e. not structs or unions
  - and not arrays (although individual elements of arrays of primitive elements are fine)
- Atomic behaviour is usually only supported for a single instruction
  - Not a sequence of instructions

Solutions

```
// transfer 10000 from account a to b

balance_a -= 10000
                     <---- interrupted here
balance_b += 10000
```

- Prevent interruptions?
- Prevent other processes from reading or writing `balance_a` and `balance_b` until transfer is complete

Preventing Interruptions

- Most OSes today are pre-emptive
- You cannot prevent your process from being context-switched
  - in general, at least

Mutual Exclusion

- We want to allow only one process to read/write `balance_a` and `balance_b`
- This is the problem of *mutual exclusion*

Semaphores

- Semaphores are a general solution to the mutual exclusion problem
  - Other more efficient mechanisms exist
- A semaphore is an object that supports two operations
  - "wait" and "signal"
  - historically called "P" and "V"
- POSIX supports semaphores

Using a Named Semaphore

```
/* create a semaphore with value 1 */
sbalance = sem_open("/balance", O_CREAT, S_IRUSR | S_IWUSR, 1);
if(sbalance == SEM_FAILED) {
  perror("sem_open");
  exit(1);
}

printf("waiting to enter\n");

/* waits if current value of semaphore is == 0, otherwise decrements
   it and returns immediately */
while(sem_wait(sbalance) != 0);

printf("in critical section\n");
balance_a -= amount;
balance_b += amount;

/* increases semaphore value by 1, releases a waiting
    process if value > 0 */
sem_post(sbalance);
printf("done\n");

sem_unlink("/balance");
```

Explanation

- Create a named semaphore
  - The name appears in the filesystem, so different processes can share the same semaphore
- Initialize it with value 1
  - To indicate only one process can read/write balances
- Each process then calls `sem_wait` before reading/writing balances
  - This will force the process to wait if another process is already reading/writing balances (i.e. the semaphore value will be 0 or less)
- The process that is in the critical section should call `sem_post` when it is done
  - `sem_post` increases the semaphore value
  - This allows another process to enter the critical section

Other IPC mechanisms

- Shared memory and ordering are the basic building blocks
- Other inter-process communication mechanisms exist:
  - SystemV shared memory (for systems that don't support MAP_ANONYMOUS),
  - Pipes (one-way communication between programs),
  - Message queues,
  - Sockets (see textbook, if interested, or take CSC2/457)
  - and other Linux-specific mechanisms

References

- Chapter 12 of the textbook
  - Same issues as presented here, but with different examples
  - Also uses unnamed semaphores
  - Also focuses more on thread-based concurrency, which we'll discuss later
  - Not up to date with the latest in C11
- Start from Overview Manual page for semaphores
- No good overview of atomic variables I could find yet
  - The C11 Standard details their behaviour, but it's not introductory