Operating Systems - Spring 2009
Assignment 2: Process Synchronization

Karl Marklund <karl.marklund@it.uu.se>
Assume we have one shared bank account.

All clients who has access to the account can make deposits.
We model the bank account using *shared memory*.

We model each client as a separate Unix *process*.

Using `fork()` each process uses separate memory...

In the lab you will learn how to create a shared memory segment.
deposit(int ammount) {
    balance = balance + ammount;
}

int balance = 0;

Client A
deposit(100);

Client B
deposit(50);

Remember: Incrementing the balance is not performed as one atomic operation by the hardware...

One possible scenario.

<table>
<thead>
<tr>
<th></th>
<th>Client A</th>
<th>Client B</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load (balance)</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Add (balance, 100)</td>
<td>0</td>
<td>0</td>
<td>150</td>
</tr>
<tr>
<td>Store (balance)</td>
<td>100</td>
<td>0</td>
<td>150</td>
</tr>
<tr>
<td>Load (balance)</td>
<td>100</td>
<td>0</td>
<td>150</td>
</tr>
<tr>
<td>Add (balance, 50)</td>
<td></td>
<td>100</td>
<td>150</td>
</tr>
<tr>
<td>Store (balance)</td>
<td></td>
<td>100</td>
<td>150</td>
</tr>
</tbody>
</table>
Another possible scenario.

Oops! A *race condition* on the shared variable balance.

<table>
<thead>
<tr>
<th>Client A</th>
<th>Client B</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load (balance)</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Add (balance, 50)</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Store (balance)</td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>Load (balance)</td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>Add (balance, 100)</td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>Store (balance)</td>
<td></td>
<td>150</td>
</tr>
</tbody>
</table>
Must make sure only one process updates the account at the time...

Keep exactly one cookie in a jar.

Processes can look in the jar and *grap* the cookie if available.

If the cookie is absent – wait until another process *puts* it back.
Account

int balance = 0;

Client A
Grap(cookie);
deposit(100);
Put(cookie);

Client B
Grap(cookie);
deposit(50);
Put(cookie);
int balance = 0;

Client A
Grap(cookie);
deposit(100);
Put(cookie);

Grab is called wait.

Put is called signal.

Client B
Grap(cookie);
deposit(50);
Put(cookie);

The cookie jar is called a binary semaphore.
More than one "cookie" in the jar.

*Signal*: increments the semaphore counter.

*Wait*: if counter > 0, decrement counter, otherwise wait.
In the lab you will learn how to use semaphores in C to synchronize processes.
#include "semaphore.h"

int n = 3;
Semaphore sem = Semaphore_create(n);
Semaphore_wait(sem);
Semaphore_signal(sem);
Semaphore_destroy(sem);

Simplified API to SystemV Semaphores.

Create a new counting semaphore, initialize counter to n.

NOTE: you must destroy each semaphore you create – they will not be deleted when your program/process terminates.
In the lab you will learn how to use shared memory to share data between processes.
#include "shared_memory.h"

// Declare a handle to a shared memory segment.
Shared_memory shmid;

// Declare a pointer to data to be shared.
int *balance;

// Create the shared memory segment.
schmid = Shared_memory_create(sizeof(int));

// Must attach the segment to the process address space.
balance = (int*) Shared_memory_attach(shmid);

// Initialize data.
*balance = 1000;

NOTE: balance is a pointer to an int. Must use the dereference operator * to read/write the data stored at the pointer address.
// Don't forget to detach the segment from the
// process address space when the process
// is done using the segment.
Shared_memory_detach(balance);

// Don't forget to destroy the shared segment
// when all processes sharing the segment are
// done using the segment.
Shared_memory_destroy(shmid);

NOTE: Shared memory segment will not be deleted automatically
when a process exit.