Exec

- Replaces the current process, does not create a new process
  - Commonly used with fork. Fork first creates a new process and then exec loads a program and has the newly created process run it.
- Many uses for exec, for example the shell uses fork and exec to run commands.

Note: Example code is from Hal Perkin’s 333 course. Thanks to Hal and his team for the shell code.
x86-64 Calling Conventions

- `%rdi`  
  - Holds the first argument
- `%rsi`  
  - Holds the second argument
- `%rsp`  
  - Points to the top of the stack/lowest address (stack grows down)

Local variables are stored on the stack (If arguments are arrays, store them on the stack and store a pointer in the register)
int main(int argc, char *argv)

- First argument will always be `argc` (number of arguments)
- Second argument will always be `argv`, an array of strings (first string is always the name of the program)
Exec Stack Layout

- argv is an array of pointers, therefore %RSI points to an array on the stack.
- Since each element of the argv array is a char *, each element points to a string stored elsewhere on the stack.
- You can think of all variables stored above the return PC on the stack as local variables of the caller.
Let’s Practice!
(Get out some paper and pens!)
Practice Exercise 1 - “cat cat.txt”

TODO:
Draw out the stack layout and determine the register values for exec called with “cat cat.txt”.

<table>
<thead>
<tr>
<th>Registers</th>
<th>Stack grows down</th>
</tr>
</thead>
<tbody>
<tr>
<td>%RDI</td>
<td>?</td>
</tr>
<tr>
<td>%RSI</td>
<td>?</td>
</tr>
<tr>
<td>%RSP</td>
<td>?</td>
</tr>
</tbody>
</table>

Registers

High addresses

Low addresses
Practice Exercise 1 - “cat cat.txt” Solution

- %RDI, the first argument, holds argc, which is 2.
- %RSI, the second argument, holds argv, which is a pointer to the beginning of the argv array.
- %RSP, the stack pointer, has been properly adjusted to point to the bottom of the stack. The value of the return PC does not matter.
Practice Exercise 2 - “kill -9 500”

TODO:
Draw out the stack layout and determine the register values for exec called with “kill -9 500”.

Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%RDI</td>
<td>?</td>
</tr>
<tr>
<td>%RSI</td>
<td>?</td>
</tr>
<tr>
<td>%RSP</td>
<td>?</td>
</tr>
</tbody>
</table>
%RDI, the first argument, holds argc, which is 3.
%RSI, the second argument, holds argv, which is a pointer to the beginning of the argv array.
%RSP, the stack pointer, has been properly adjusted to point to the bottom of the stack. The value of the return PC does not matter.
Pipes

- Pipes are a mechanism used for inter-process communication (IPC).
- With the `sys_pipe`, a process sets up a writing and reading end to a “holding area” where data can be passed from process to process.
- What should happen if the write end or the read end is closed (by potentially multiple writers/readers)? When can you free the buffer of the pipe?
Pipe allocation

- Pipes should be allocated at runtime, when the pipe is requested by a process
  - What mechanism does xk provide to allocate memory dynamically?
- Each pipe should behave like a file so that the file-oriented system calls can work as normal with the pipe
  - How can you determine whether a struct file is an inode or a pipe?