Processes

Dr. Sam Hsu
Computer Science & Engineering
Florida Atlantic University
Typical Memory Layout of a Process (1/3)

- **Text segment**
  - For instructions.
- **Data segment**
  - Initialized data segment.
  - Uninitialized data segment.
    - Bss (block started by symbol)
- **Stack**
  - For function calls.
Typical Memory Layout of a Process (2/3)

- Heap
  - For dynamic memory allocations.
- Command-line arguments
  
  `argc/argv[]/envp[]`
- Environment variables
  
  `extern char **environ`
Typical Memory Layout of a Process (3/3)

- **High address**
  - argc/argv[]
  - Stack
  - Heap
    - Uninitialized data
    - Initialized data
    - Text

- **Low address**
  - Text
  - Initialized to zero by `exec()`
  - Read from program by `exec()`

- **Command-line arguments and environment variables**
A process is a program in execution. Each has a unique PID. A non-negative integer: 0 ~ PID_MAX. Created by `fork()`/`vfork()` system calls. Some special PIDs: 0: scheduler 1: init 2: pagedaemon
The `fork()` System Call (1/3)

- Only way to create processes
  - Except for 0, 1, ...
- Parent/child relationship
  - The child is a copy of the parent.
    - It inherits the parent's data, heap and stack.
  - COW (copy-on-write) in most current implementations.
    - Only the page that gets modified is copied, typically in a virtual memory system.
The `fork()` System Call (2/3)

- Often the parent and the child share the text segment,
  - If it is read-only.
- Never know whether the parent or child will start executing first.
  - All file descriptors that are open in the parent are duplicated in the child.
  - Parent/child also share the same file offset (Files opened after `fork()` are not shared).
The \textit{fork()} System Call (3/3)

- Two normal cases for handling the descriptors after a \textit{fork()}:
  - Parent waits.
  - Parent and child go their own way.

- \textit{fork()} may fail if it,
  - Exceeds user limit, or
  - Exceeds total system limit.

- Two uses (reasons) for \textit{fork()}:
  - Each can execute a different sections of the code at the same time.
  - One process can execute a different program.
The *vfork()* System Call

- A BSD variant of *fork()*; now supported by SVR4.
- Similar to *fork()*; however, is used to exec a new program only.
- Child running in the parent address space until it calls *exec()*/*exit()*.
- Not fully copying the address space of the parent into the child.
- *vfork()* guarantees that the child runs first until it calls *exec()*/*exit()*.
- Deadlock is possible if the child needs information from the parent.
Process Termination

- Normal termination
  - Return from main().
  - Calling `exit()`.
  - Calling `_exit()`.

- Abnormal termination
  - Calling `abort()`.
  - Terminated by a signal.
The `exit()`/`_exit()` System Calls

- **exit()**
  - Performs a standard I/O cleanup.
    - Executes all registered exit handlers.
    - Flushes all C output buffers.
    - Closes all open streams.
  - Terminates the calling process.

- **_exit()**
  - Terminates the calling process without performing a standard I/O cleanup.
Various \textit{wait()} System Calls (1/3)

- \textit{wait()} is used to wait for a child to terminate.
- \textit{waitpid()} is used to wait for a specific child to terminate, plus some options.
- \textit{wait3()/wait4()} will further collect resource usage information.
Various *wait()* System Calls (2/3)

- When a process terminates, the following are reported/returned to its parent:
  - Exit status.
  - Some timing statistics (CPU time consumed).
  - Etc.
Various *wait()* System Calls (3/3)

<table>
<thead>
<tr>
<th>Function</th>
<th>pid</th>
<th>options</th>
<th>rusage</th>
<th>POSIX.1</th>
<th>SVR4</th>
<th>4.4BSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait()</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>waitpid()</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>wait3()</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>wait4()</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>

Arguments supported by various *wait()* functions on different systems.
Zombie Process

- A process that no longer exists, but still ties up a slot in the system process table.
- A process that has terminated, but whose parent exists and has not waited or acknowledged the child's termination.
- Note: In SVR4, no zombie will be created if the parent selects to ignore the termination of a child process. Its information is thus discarded.
Orphaned Process (orphan)

- A process whose parent has exited.
- An orphaned process can never become a zombie process.
- Its slot in the process table is immediately released when an orphan terminates.
- Orphaned processes are inherited by init().
Race Conditions

- A race condition occurs when multiple processes are competing for the same system resource(s).
  - The final outcome depends on the order in which the processes run.
- Problems due to race conditions are hard to debug.
  - Programs tend to work “most of the time.”
- Needs to have process synchronization.
A process has the following IDs:

- Process ID.
- Parent Process ID.
- Process group ID.
- Session ID.
- User ID of the process.
- Group ID of the process.
- Effective user ID.
- Effective group ID.
Process Attributes (2/2)

- Some other properties:
  - Controlling terminal.
  - Current working directory.
  - Root directory.
  - Open files descriptors.
  - File mode creation mask.
  - Resource limits.
  - Process times.
Two Kernel Data Structures Pertinent to a Process

- The process table entry and user (u) area.
  - They contain administrative information for a process.
  - One each per process.

- Process table entry
  - It keeps information always needed.

- User area
  - It keeps information needed when running.
The Context of a Process

- User address space.
- Relevant kernel data structures:
  - Process table entry + u area.
- Contents in hardware registers.
The `exec()` System Call (1/5)

- Only way to execute processes.
  
  In the UNIX system, `fork()` creates processes and `exec()` executes processes. These two system calls are very closely related. Without `exec()`, no process can be executed. No `fork()`, no process can be created. They make a good team achieving most of the UNIX system operations.

- Will replace the calling process with a new program and start execution.
The *exec()* System Call (2/5)

- Brand new text, data, heap and stack segments.
- Inherits most of the process attributes of the calling process, such as
  - PID and PPID.
  - The real and effective UID and GID that aren’t SUID or SGID.
  - Open files, except those with the close-on-exec flag set, are passed to the new program.
  - The file mode creation mask (umask) is passed to the new program.
The `exec()` System Call (3/5)

- Controlling terminal.
- Current working directory
- Root directory.
- File locks.
- Signal mask.
- Pending signals.
- Resource limits
- CPU times.
The **exec()** System Call (4/5)

- Is a family name for six like functions virtually doing the same thing, only slightly different in syntax:
  - `exec()`, `execv()`, `execl()`, `execve()`, `execlp()`, and `execvp()`.
  - Only `execve()` is a system call.
- Meaning of different letters:
  - **l**: needs a list of arguments.
  - **v**: needs an `argv[]` vector (l and v are mutually exclusive).
  - **e**: needs an `envp[]` array.
  - **p**: needs the PATH variable to find the executable file.
The `exec()` System Call (5/5)

Relationship of the `exec()` functions.