Fork() System Call and Processes

CS449 Fall 2017
Programs and Processes

- **Program**: Executable binary (code and static data)
- **Process**: A program loaded into memory
  - Program (executable binary with data and text section)
  - Execution state (heap, stack, and processor registers)
  - OS state (open file descriptors, page table, etc.)

```c
int foo() {
    return 0;
}

int main() {
    foo();
    return 0;
}
```
fork(): How Processes are Born

• **fork():** A system call that creates a new process identical to the calling one
  – Makes a copy of text, data, stack, and heap
  – Starts execution on that new copy in **parallel** with old copy
  – Old copy: **parent** process, new copy: **child** process
Before fork()

Parent (original state)

Child (copy of state)

After fork()
fork(): How Processes are Born

- **fork():** A system call that creates a new process identical to the calling one
  - Makes a copy of text, data, stack, and heap
  - Starts execution on that new copy in parallel with old copy
  - Old copy: parent process, new copy: child process
  - Returns: child PID for parent process, 0 for child process
  - Used as the point of divergence between parent and child

- **Uses of fork()**
  - To create a parallel program that can execute on multiple CPUs
    - E.g. Web server may fork a process on each webpage request
  - To launch a new program
    - E.g. thoth $ ls ➔ Linux shell process forks off ‘ls’ process
fork() example

```c
#include <stdio.h>
#include <unistd.h>

int main()
{
    int seq = 0;
    if(fork()==0)
    {
        printf("Child! Seq=%d\n", ++seq);
    }
    else
    {
        printf("Parent! Seq=%d\n", ++seq);
    }

    printf("Both! Seq=%d\n", ++seq);
    return 0;
}
```

```
>> ./a.out
Parent! Seq=1
Both! Seq=2
Child! Seq=1
Both! Seq=2

>> ./a.out
Child! Seq=1
Both! Seq=2
Parent! Seq=1
Both! Seq=2

>> ./a.out
Parent! Seq=1
Child! Seq=1
Both! Seq=2
Both! Seq=2
```
fork() example

```c
#include <stdio.h>
#include <unistd.h>
int main()
{
    int seq = 0;
    if(fork()==0)
    {
        printf("Child! Seq=%d\n", ++seq);
    }
    else
    {
        printf("Parent! Seq=%d\n", ++seq);
    }
    printf("Both! Seq=%d\n", ++seq);
    return 0;
}
```

- Differentiate child and parent using return value of fork():
  - For Child: 0
  - For Parent: child’s process id
- **Copies** execution state (not shares)
  - Child has its own stack with own copy of seq variable
  - Updates to seq in child not reflected on parent (vice versa)
Copy-On-Write: How fork() is Efficient

• Inefficient to immediately copy physical memory
  – Code (text section) remains identical after fork
  – Even portions of data, heap, and stack may remain identical

• Copy-On-Write
  – OS policy to lazily copy pages only when they are modified
  1. Initially map same physical page to child virtual memory space
     • But marking the page read-only in each page table
  2. Write to shared physical page triggers page fault exception
  3. OS handles exception by making a physical copy of page and remapping the written virtual page to the new page
     • New page will now be marked as writable
  – OS keeps track of shared physical pages using a per-page reference counter of how many virtual pages share that page
How is the First Process Born?

• Who is the very first ancestor (the “Adam”)?
• init process
  – Created by the OS after system boot up
  – Very first process: has process id (PID) of 1
  – Reads and executes a system initialization script in a standard location (e.g. /etc/rc)
    • Typically forks off a bunch of background “server” processes (e.g. SSH server, web server, file server, etc.)
  – All processes are descended from init
Multitasking: Concurrently Running Processes on One Processor

Single $EIP (CPU’s point of view)

Multiple $EIPs (process point of view)

Isolated address spaces that are not interrupted
Context Switch: Switching Tasks

- OS provides the illusion of a dedicated CPU per process
- By alternating CPU between executing different processes
- **Context switch**: Switching between two processes
  - Saves “context” of one process, restoring that of another one
  - Period given to each process is called a **time slot**
Process Context

• Context: state that must be saved by a task on interruption and restored on continuation
  – Consists of: Data in registers + OS state
  – Does not include data in memory
    • Code, data, stack, heap are not part of context
    • Physical memory is spatially shared by multiple processes

• OS state contained in data structure called the **process control block (PCB)**
  – Process ID, or PID (unique identifier for process)
  – Memory management info (pointer to page table etc.)
  – I/O status info (pointer to file descriptor table etc.)
  – Process scheduling info (scheduling priority etc.)
Dispatch Mechanism

- **OS maintains list of all processes (PCBs)**
- **Each process is in one of 3 modes:**
  - **Running:** Executing on the CPU
  - **Ready:** Waiting to execute on CPU
  - **Blocked:** Waiting for I/O or synchronization
- **Dispatch Loop:** on all non-blocked processes

```c
while (1) {
    run process for a while;
    stop process and save its state;
    load state of another ready process;
}
```
Life Cycle of a Process

- Created
- Ready
- Blocked (waiting)
- Running
- Exit
When does the dispatcher run?

- Remember: Kernel is a reactive program
  - Only runs in response to user process request or HW event
  - Cannot rely on user process to request dispatcher to run
  - Processes can be greedy or worse malicious

- How does the kernel wrest control from a process?
  - Opportunistic method: Run dispatcher while handling...
    - Exceptions: System calls, page faults, etc
    - Interrupts: Keyboard strokes, network packet arrivals, etc.
  - Guaranteed method: Set HW timer for length of time slot
    - Run dispatcher when timer interrupt fires
  - Most kernels (including Linux) make use of both
    - Opportunistic: To maximize responsiveness
    - Timer-based: To guarantee fairness
Process / Virtual Memory Comparison

- **Process** is to **processor** what **virtual memory** is to **physical memory**

<table>
<thead>
<tr>
<th></th>
<th>Virtual Memory</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Goal</strong></td>
<td>Abstraction for physical memory</td>
<td>Abstraction for processors</td>
</tr>
<tr>
<td></td>
<td>Make application oblivious of the physical memory</td>
<td>Make application oblivious of the physical processors</td>
</tr>
<tr>
<td></td>
<td>underneath</td>
<td>underneath</td>
</tr>
<tr>
<td><strong>Sharing</strong></td>
<td>Spatial: Multiple virtual memory spaces on one physical memory</td>
<td>Temporal: Multiple processes alternate on one processor</td>
</tr>
<tr>
<td><strong>Resource Limitation</strong></td>
<td>Swaps pages in &amp; out of hard disk when memory is too small</td>
<td>Swaps processes in &amp; out of CPUs when too few CPUs</td>
</tr>
<tr>
<td><strong>OS Control</strong></td>
<td>OS controls mapping from virtual memory to physical memory</td>
<td>OS controls mapping from processes to processors</td>
</tr>
</tbody>
</table>
Using fork() to Launch New Program

• **Sequence of fork() and execvp()**
  1. fork(): Clone current process
  2. execvp(): overwrite current process with new program

• **execvp(char *filename, char **argv)**
  – Starts execution of program given by filename
  – Invokes dynamic linker on filename, start from scratch
    1. Loads in text and data sections of new executable image
    2. Loads in any shared objects and performs dynamic linking
    3. Starts executing from entry point given by executable header
  – What Linux shell calls when launching a program after forking
# execvp() example

```c
#include <stdio.h>
#include <unistd.h>
int main()
{
    if(fork()==0)
    {
        execvp(args[0], args);
        // DOES NOT GET HERE
        printf("Child!\n");
    }
    else
    {
        printf("Parent!\n");
    }
    return 0;
}
```

```
>> ./a.out
Parent!
drwx------ 4 wahn UNKNOWN1 4096 Oct 21 08:13 .
drwxr-xr-x 10 wahn UNKNOWN1 2048 Oct 21 08:13 ..
-rwxr-xr-x 1 wahn UNKNOWN1 6743 Oct 21 08:12 a.out
```

- execvp never returns since memory is overwritten using another program image (ls) by dynamic linker
Managing processes (Unix)

• Finding processes
  – ‘ps’, ‘pstree’

• Monitoring Processes
  – ‘top’

• Stopping processes
  – ‘kill <pid>’ (for a soft kill using SIGTERM)
  – ‘kill -9 <pid>’ (for a hard kill using SIGKILL)

• Procfs (/proc/)
Using ‘ps’

- Listing processes associated with this terminal

<table>
<thead>
<tr>
<th>UID</th>
<th>PID</th>
<th>PPID</th>
<th>C</th>
<th>STIME</th>
<th>TTY</th>
<th>TIME</th>
<th>CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>wahn</td>
<td>11848</td>
<td>11847</td>
<td>0</td>
<td>06:27</td>
<td>pts/1</td>
<td>00:00:00</td>
<td>-bash</td>
</tr>
<tr>
<td>wahn</td>
<td>15754</td>
<td>11848</td>
<td>0</td>
<td>11:24</td>
<td>pts/1</td>
<td>00:00:00</td>
<td>ps -f</td>
</tr>
</tbody>
</table>

- Listing all processes

<table>
<thead>
<tr>
<th>UID</th>
<th>PID</th>
<th>PPID</th>
<th>C</th>
<th>STIME</th>
<th>TTY</th>
<th>TIME</th>
<th>CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Aug26</td>
<td>?</td>
<td>00:00:11</td>
<td>/sbin/init</td>
</tr>
<tr>
<td>root</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>Aug26</td>
<td>?</td>
<td>00:00:00</td>
<td>[kthreadd]</td>
</tr>
<tr>
<td>root</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>Aug26</td>
<td>?</td>
<td>00:00:03</td>
<td>[migration/0]</td>
</tr>
<tr>
<td>root</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>Aug26</td>
<td>?</td>
<td>00:00:04</td>
<td>[ksoftirqd/0]</td>
</tr>
<tr>
<td>root</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>Aug26</td>
<td>?</td>
<td>00:00:00</td>
<td>[migration/0]</td>
</tr>
</tbody>
</table>
Using ‘pstree’

- Displays all processes in the form of a tree

```
thoth $ pstree -p
init(1)─┬─NetworkManager(1493)
    ├─abrt-dump-oops(2011)
    └─abrt(2001)
    └─acpid(1681)
         .
         .
         .
    └─sshd(20847)─┬─sshd(10995)─sshd(11007)─bash(11008)─nano(11144)
        └─sshd(11125)─sshd(11847)─bash(11848)─pstree(15369)
```

- Note how all process are forked off the ‘init’ process
Using ‘kill’

• Killing your own shell

```bash
thoth $ ps -f
UID   PID   PPID  C   STIME   TTY      TIME   CMD
wahn  11848 11847 0  06:27 pts/1   00:00:00 -bash
wahn  15904 11848 0  11:36 pts/1   00:00:00 ps -f
thoth $ kill 11848
thoth $ kill -9 11848
Connection to thot.cs.pitt.edu closed.
```
Procfs

• Pseudo file system for special ‘process’ files
  – Typically mounted on /proc/
  – Organized by PID
  – /proc/self contains information about current process

• Information available
  – Open file descriptors
  – Virtual memory map
  – Cmdline
  – State of process (running / ready / blocked)
  – Etc...
Procfs usage examples

- Listing open file descriptors for process

```bash
thoth $ ls -l /proc/self/fd
total 0
lrwx------ 1 wahn UNKNOWN1 64 Sep 11 13:37 0 -> /dev/pts/0
lrwx------ 1 wahn UNKNOWN1 64 Sep 11 13:37 1 -> /dev/pts/0
lrwx------ 1 wahn UNKNOWN1 64 Sep 11 13:37 2 -> /dev/pts/0
lr-x------ 1 wahn UNKNOWN1 64 Sep 11 13:37 3 -> /proc/16970/fd
```

- Showing virtual memory map

```bash
thoth $ cat /proc/self/maps
00400000-0040b000 r-xp 00000000 fd:00 2681 /bin/cat
0080a000-0080b000 rw-p 0000a000 fd:00 2681 /bin/cat
0080b000-0082c000 rw-p 00000000 00:00 0 [heap]
34ff600000-34ff78b000 r-xp 00000000 fd:00 131 /lib64/libc-2.12.so
34ff98e000-34ff98f000 rw-p 0018e000 fd:00 131 /lib64/libc-2.12.so
7fffffff0ea000-7fffffff000 rw-p 00000000 00:00 0 [stack]
```