Process Abstraction

Nima Honarmand

Administrivia

- **Course staff email**: cse306ta at cs.stonybrook.edu
  - Both Babak and I will be monitoring the account to ensure a timely response
- **What to use it for**: any email that you would otherwise send to my or the TA's email
  - Unless it is for my eyes only
- Remember to use the **Blackboard forum** for all non-private questions or class/lab-related discussions
- Check your **CS email account** for your VM addr. and key
  - Why not have all your emails forwarded to one account?

What is a Process?

- **Process**: *dynamic* instance of a program

vs.

- **Program**: *static* code and data

- What does a process consist of?
  - Abstraction of CPU: *threads*
  - Abstraction of memory: *address space*
  - Abstraction of devices: *file handles* (for storage), *sockets* (for NIC), etc.

What is a Process?

- **Process** = **Program** (static code and data) + **execution state**
- Execution state consists of
  - Thread context: General purpose registers, stack pointer, program counter, etc.
  - Address space content: code, stack, heap, memory-mapped files
  - Open files, sockets, etc.
- **Program** is used to initialize the **execution state** which then changes as program executes
- The OS keeps track of each process' execution state in a data structure called *Process Control Block* (*PCB*)

Program to Process

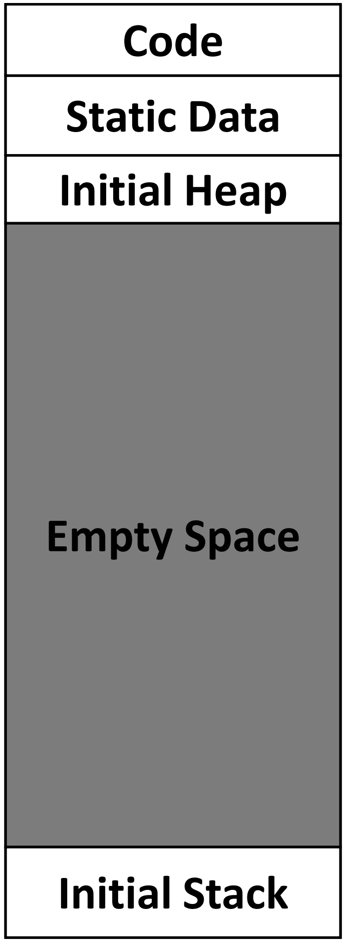
- We write a program in, e.g., C++
- A compiler translates that program into a binary containing
  - Headers (e.g., address of first instruction to execute)
  - Code section (.text, .init, .plt)
  - Data sections (.data, .bss, .rodata, .got, etc.)
  - And other sections we don’t care about now
- OS creates a new process and uses the program to initialize its state

Initializing Process State

- Initialize address space
  - Load code and data into memory
  - Setup a piece of memory for initial stack (including space for command line arguments and environment variables)
  - Setup a piece of memory for the initial heap
  - etc.

- Initialize the first thread
  - Initialize (zero-out) the general purpose registers
  - Set the program counter to the first instruction
  - Set the stack pointer to the top of stack
  - etc.

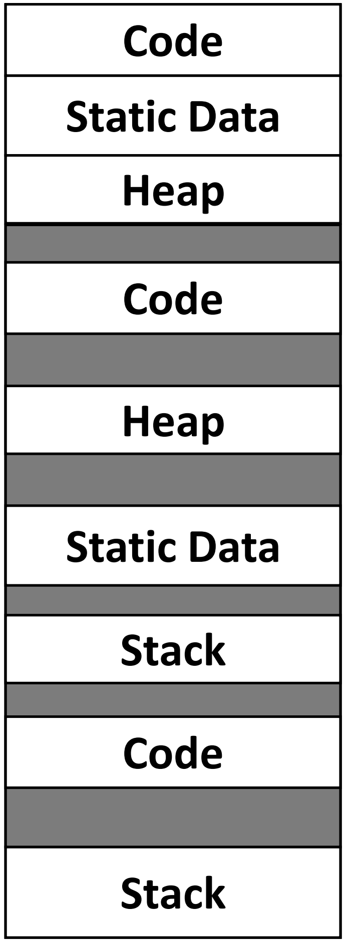
Example of Initial Address Space



Changing Process State

- As the process runs, this layout changes
  - Might need more heap space
  - Might become multi-threaded and more need more stacks
  - Stacks might grow
  - Might load more code and more static data
  - etc.

Example of Running Address Space



Virtualizing the CPU

- Many more threads (abstract CPUs) than physical CPUs
- Have to multiplex threads over CPUs
- Key technique: Context Switching
  - Thread A runs for some time, then we switch to thread B, and so on
  - Temporal Multiplexing of CPU: different threads occupy the same CPU at different points of time
- How to switch context? Save A's register to its PCB, restore B's register from its PCB
- When to switch context? We'll see in future lectures

Virtualizing the Memory

- Many process address spaces and only one physical memory space
- Have to multiplex again
- Key technique: Virtual Memory
  - Addresses generated by each process are relative to its own address space
  - They pass an OS-controlled translation layer before being sent to memory
  - Spatial Multiplexing of memory: different address spaces reside at different parts of physical memory simultaneously

Isolation and High Performance

- Need for Isolation
  - Processes should be isolated (protected) from each other
  - OS kernel should be isolated (protected) from processes
  - Hardware devices should be protected from processes
- We also want high performance
  - Applications should execute directly on the processor
  - I.e., the OS should not need to intervene and check the validity of every single instruction the application wants to execute
- How to provide isolation and high performance simultaneously?
- Answer: Limited Direct Execution (LDE)

Limited Direct Execution (LDE)

- Two important hardware features to enable LDE:
  1) Separate user/supervisor modes for the processor
  2) Virtual Memory Hardware (*a.k.a.* Memory Management Unit or MMU)

- User (non-privileged) mode
  - Only a subset of "harmless" processor instructions are available
    - Arithmetic and logic operations, branches, memory load/store
    - Only a few general-purpose registers accessible
- Supervisor (privileged) mode
  - All processor instructions are available including control instructions
    - E.g., enable/disable interrupts, change the page table, performance counters, …
    - All general-purpose as well as control registers are accessible

LDE: Separate Modes

- Applications exclusively run in the non-privileged mode
  - Can do whatever permitted in that mode without OS intervention
    - Change register values, read/write their own stack or heap, do ALU operations, take branches, call functions in their code segment, etc.
  - Anything else requires switching to privileged mode (i.e., making a syscall) at which point the kernel takes over

→ Applications execute directly on the processor but are limited to what's available in the non-privileged mode

- But how is this mode transfer (user-to-supervisor and vice versa) implemented?
  - Answer: interrupts (next lecture)

LDE: Virtual Memory

- Someone has to make sure processes only access their own memory. But who?
  - OS cannot check every single memory access a process performs. Would be too slow.
  - Hardware (processor) has to do it directly

- But how does the processor know which memory accesses are valid for a given process?
  - The OS tells it by setting up the MMU when switching to a process
  - Review: Page Table, TLB (Translation Lookaside Buffer)

→ So a process can access its memory directly as long as it respects the MMU limitations

Process API Recap

Where New Processes Come From

- Parent/child model
- An existing program has to spawn a new one
  - Most OSes have a special `init` program that launches system services, logon daemons, etc.
  - When you log in (via a terminal or SSH), the login program spawns your shell

Approach 1: Windows `CreateProcess()`

- In Windows, when you create a new process, you specify a new program
  - And can optionally allow the child to inherit some resources (e.g., an open file handle)

Approach 2: Unix `fork/exec`

- In Unix, a parent makes a copy of itself using `fork()`
  - Child inherits everything, runs same program
  - Only difference is the return value from fork()
- A separate `exec()` system call loads a new program
- Major design trade-off:
  - How easy to inherit
  - vs. Security (accidentally inheriting something the parent didn't intend)
  - Note that security is a newer concern, and Windows is a newer design…

Why Separate `fork/exec`

- Life with `CreateProcess(filename);`
  - But I want to close a file in the child.
    `CreateProcess(filename, list of files);`
  - And I want to change the child's environment.
    `CreateProcess(filename, CLOSE_FD, new_envp);`
  - Etc. (a very ugly etc.)

```
BOOL WINAPI CreateProcess(
  _In_opt_     LPCTSTR               lpApplicationName,
  _Inout_opt_  LPTSTR                lpCommandLine,
  _In_opt_     LPSECURITY_ATTRIBUTES lpProcessAttributes,
  _In_opt_     LPSECURITY_ATTRIBUTES lpThreadAttributes,
  _In_         BOOL                  bInheritHandles,
  _In_         DWORD                 dwCreationFlags,
  _In_opt_     LPVOID                lpEnvironment,
  _In_opt_     LPCTSTR               lpCurrentDirectory,
  _In_         LPSTARTUPINFO         lpStartupInfo,
  _Out_        LPPROCESS_INFORMATION lpProcessInformation
);
```

Why Separate `fork/exec`

- **`fork()`** = split this process into 2 (new PID)
  - Returns 0 in child
  - Returns pid of child in parent
- **`exec()`** = overlay this process with new program
  - (PID does not change)

Why Separate fork/exec

- Let you do anything to the child's process environment without adding it to the `CreateProcess()` API.

```
int pid = fork();                // create a child
if (0 == pid) {                  // child continues here
   // Do anything (unmap memory, close net
   // connections…)
   exec("program", argc, argv0, argv1, …);
}
```

- `fork()` creates a child process that inherits:
  - identical copy of all parent's variables & memory
  - identical copy of all parent's CPU registers (except one)
- Parent and child execute at the same point after `fork()` returns:
  - for the child, `fork()` returns 0
  - for the parent, `fork()` returns the process identifier of the child

Program Loading: `exec()`

- The `exec()` call allows a process to “load” a different program and start execution at main (actually `_start`).
- It allows a process to specify the number of arguments (`argc`) and the string argument array (`argv`).
- If the call is successful
  - it is the same process …
  - but it runs a different program !!
- Code, stack & heap is overwritten
  - Sometimes memory mapped files are preserved.
- `exec()` does not return!

General Purpose Process Creation

In the parent process:

```
main()
  …
  int pid =fork();                    // create a child
  if (0 == pid) {                     // child continues here
       exec_status = exec("calc", argc, argv0, argv1, …);
       printf("Something is horribly wrong\n");
       exit(exec_status);
} else {                              // parent c   tinues here
  printf("Who's your daddy?");
  child_status = wait(pid);
}
```

`exec()` should not return

Exmpl: Shell forks & executes calc

```
int pid = fork();
if(pid == 0) {
 close(".history");
 exec("/bin/calc");
} else {
 wait(pid);
```

```
int pid = fork();
if(pid == 0) {
 close(".history");
 exec("/bin/calc");
} else {
 wait(pid);
```

USER

OS

pid = 127
open files = ".history"
last_cpu = 0

pid = 128
open files =
last_cpu = 0

Process Control Blocks (PCBs)

Exmpl: Shell forks & executes calc

```
int pid = fork();
if(pid == 0) {
 close(".history");
 exec("/bin/calc");
} else {
 wait(pid);
```

```
int calc_main(){
   int q = 7;
   do_init();
   ln = get_input();
   exec_in(ln);
```

USER

OS

pid = 127
open files = ".history"
last_cpu = 0

pid = 128
open files =
last_cpu = 0

Process Control Blocks (PCBs)

Cost of `fork()`

- Simple implementation of `fork()`:
  - allocate memory for the child process
  - copy parent's memory and CPU registers to child's
  - *Expensive* !!
- In 99% of the time, we call `exec()` after calling `fork()`
  - the memory copying during `fork()` operation is useless
  - the child process will likely close the open files & connections
  - overhead is therefore high
- `vfork()`
  - a system call that creates a process "without" creating an identical memory image
  - child process should call `exec()` almost immediately
  - Unfortunate example of implementation influence on interface
    - Current Linux & BSD 4.4 have it for backwards compatibility
- Copy-on-write to implement fork avoids need for `vfork()`

Orderly Termination: `exit()`

- After the program finishes execution, it calls exit()
- This system call:
  - takes the "result" of the program as an argument
  - closes all open files, connections, etc.
  - deallocates memory
  - deallocates most of the OS structures supporting the process
  - checks if parent is alive:
    - If so, it holds the result value until parent requests it; in this case, process does not really die, but it enters the zombie/defunct state
    - If not, it deallocates all data structures, the process is dead
  - cleans up all waiting zombies
- Process termination is the ultimate garbage collection (resource reclamation).

`wait()` System Call

- A child program returns a value to the parent, so the parent must arrange to receive that value
- The `wait()` system call serves this purpose
  - Puts the parent to sleep waiting for a child's result
  - When child calls `exit()`, OS unblocks the parent and returns value passed by `exit()` along with the child pid
  - If there are no children alive, `wait()` returns immediately
  - If there are zombies waiting for their parents, `wait()` returns one of the values immediately (and deallocates the zombie)