Karlsruhe Institute of Technology

Betriebssysteme

03. Processes

Prof. Dr.-Ing. Frank Bellosa | WT 2016/2017

KARLSRUHE INSTITUTE OF TECHNOLOGY (KIT) – OPERATING SYSTEMS GROUP

Where we ended last lecture

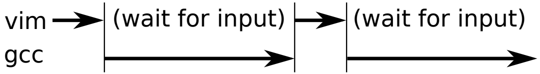
- The OS provides abstractions for and protection between application
  - Processes run without privileges in user-space
  - Kernel governs resources and runs in kernel-space
  - The distinction between kernel and user-space is made in the CPU
  - If a process executes a privileged instruction, the CPU calls the kernel instead
- Processes encapsulate all resources needed to run a program
  - Address space: all memory the process can name
  - Allocated resources, e.g., open files
- Virtual memory implements address spaces which provide protection between processes
- Processes in user-space cannot allocate resources themselves
  - The kernel provides services that perform privileged actions
  - Processes can request kernel services using system calls (syscalls)

Processes

The Process Abstraction

- Computers do “several things at the same time”
  - There are generally more such “things” than physical processors
  - It actually just looks this way → quick process switching (Multiprogramming)

- The process abstraction models this concurrency
  - Container that contains information about the execution of a program
    - Resources allocated in the OS and hardware
  - Conceptually, every process has its own “virtual CPU”
    - When switching processes, the execution context changes
    - The dispatcher switches between processes and thus between contexts
    - On a context switch, the dispatcher saves the current registers and memory mappings and restores those of the next process

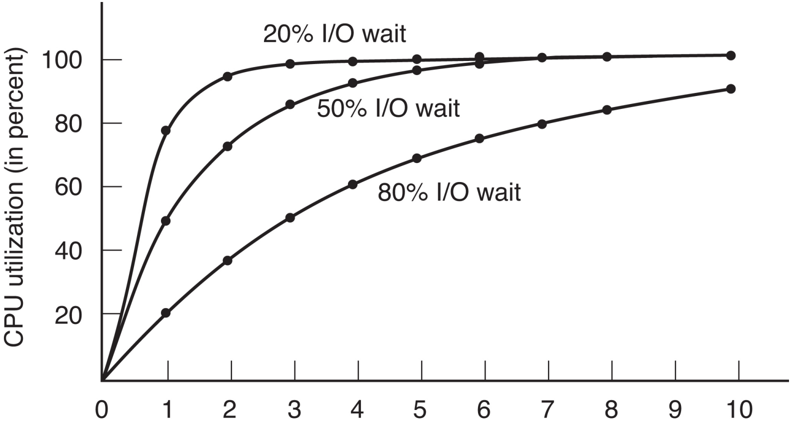


Program vs. Process is like Recipe vs. Cooking

- Recipe: Lists ingredients and gives an algorithm what to do with them
  - A program describes the memory layout and CPU instructions
- Cooking The activity of using the recipe
  - A process is the activity of executing a program
- Multiple similar recipes may exist for the same dish
  - Multiple programs may solve the same problem
- The same recipe can be cooked by several people in different kitchens at the same time
  - The same program can be run at the same time on different CPUs (as different processes)
- The same recipe can be cooked by several people at the same time
  - The same process can have several worker threads

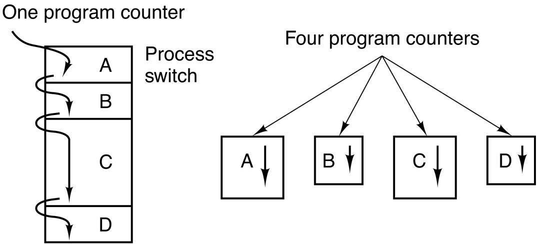
Multiprogramming can increase the CPU utilization

- With n processes suppose that a process spends fraction p of its time waiting for I/O to complete, then the CPU utilization $= 1 - p^n$



Concurrency vs. Parallelism

- The OS uses both concurrency and parallelism to implement multiprogramming
  (a) Concurrency/Pseudoparallelism: Multiple processes on the same CPU
  (b) Parallelism Processes truly running at the same time with multiple CPUs



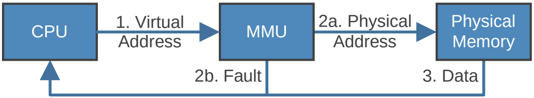
(a) (b)

- In this lecture we will focus on concurrency

Address Spaces

Virtual Memory Abstraction: Address Spaces

- Every process uses its own virtual addresses (vaddr)
  - Memory-Management Unit (MMU) relocates each load/store to physical memory (pmem)
  - Processes never see physical memory and cannot address it directly



+ MMU can enforce protection (mappings are set up in kernel mode)
  - Processes can only access what they can address (and cannot change mappings)
+ Programs can see more memory than available
  - 80:20 rule: 80% of process memory idle, 20% active working set
  - Can keep working set in RAM and rest on disk (relocate dynamically)

– Need special MMU hardware

A Process's View of the World: Address Space

- Code, data, and state need to be organized within processes resulting in an address space layout
- Generally there are three kinds of data
  1. **Fixed size** data items
  2. Data that is naturally **free'd in reverse order of allocation**
  3. Data that is **allocated and free'd dynamically** "at random"
- Compiler and architecture determine e.g., how large an integer is and what instructions are used in the text section (code)
- The loader determines based on an executable file (**`.exe`**, **`.com`**, **`ELF`**) how an **`exec`**uted program is placed in memory

1. Fixed-size Data and Code Segments

- Some data in programs never changes, other data will be written but never grows or shrinks
  - Such memory can be statically allocated when the process is created
- The BSS segment (**B**lock **S**tarted by **S**ymbol, also: .bss or bss)
  - Statically-allocated variables and variables that have not been initialized
  - The executable file typically contains the starting address and size of BSS
  - The entire segment is initially zero
- The data segment
  - Fixed-size, initialized data elements such as global variables
- The read-only data segment
  - Constant numbers and strings
- The BSS, data, and read-only data segments are sometimes summarized as a single data segment
  - Ultimately the compiler (linker) and operating system (loader) decide where to place which data and how many segments exist

2. Stack Segment

- Some data is naturally free'd in reverse order of allocation
  - **`push( a )`**
  - **`push( b )`**
  - **`pop( b )`**
  - **`pop( a )`**

- Makes memory management very easy (e.g., stack grows upwards)
  - Fixed starting point of segment (not explicitly stored in process)
  - Store top of latest allocation **`SP`** (stack pointer), initialized to starting point
  - Allocate new **`a`** byte data structure: **`SP += a; return (SP - a);`**
  - Free **`a`** byte data structure: **`SP -= a;`**

- In current CPUs, stack segment typically grows downwards!
  - Allocate: **`SP -= a; return (SP + a);`** – push CPU instruction
  - Free: **`SP += a;`** – pop CPU instruction

3. Dynamic Memory Allocation in the Heap Segment

- Some data needs to be allocated and free'd dynamically "at random"
  - E.g., input/output: don't know how large the data will be
  - Don't know how large the text document will get when starting vim

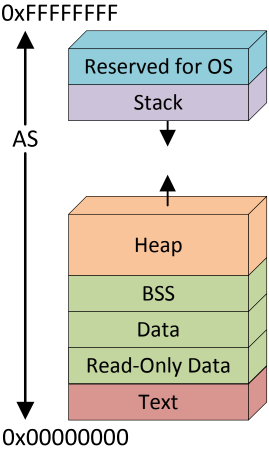
- Generally allocate memory in two tiers:

1. Allocate large chunk of memory (heap segment) from OS
   - Like stack allocation: base address + break pointer (BRK)
   - Process can get more memory from OS or give back memory by setting BRK using a system call (e.g., **`sbrk()`** in Linux)

2. Dynamically partition large chunk into smaller allocations dynamically
   - **`malloc`** and **`free`** commands that can be used in any order
   - This part happens purely in user-space!
     No need to contact kernel at this point!

Typical Process Address Space Layout

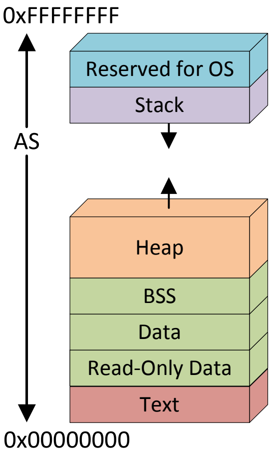
OS	Addresses where the kernel is mapped (cannot be accessed by process)
Stack	Local variables, function call parameters, return addresses
Heap	Dynamically allocated data (`malloc`)
BSS	Uninitialized local variables declared as static
Data	Initialized data, global variables
RO-Data	Read-only data, strings
Text	Program, machine code



- Instruction pointer is address in text segment
- Stack pointer is lower-most address of stack segment
- Program break pointer (BRK) is upper-most address of heap segment

Typical Process Address Space Layout

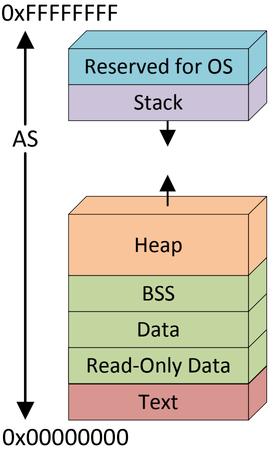
OS	Addresses where the kernel is mapped (cannot be accessed by process)
Stack	Local variables, function call parameters, return addresses
Heap	Dynamically allocated data (`malloc`)
BSS	Uninitialized local variables declared as static
Data	Initialized data, global variables
RO-Data	Read-only data, strings
Text	Program, machine code



- Instruction pointer is address in text segment
- Stack pointer is lower-most address of stack segment
- Program break pointer (BRK) is upper-most address of heap segment

Typical Process Address Space Layout

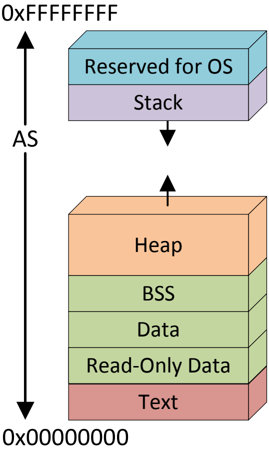
OS	Addresses where the kernel is mapped (cannot be accessed by process)
Stack	Local variables, function call parameters, return addresses
Heap	Dynamically allocated data (`malloc`)
BSS	Uninitialized local variables declared as static
Data	Initialized data, global variables
RO-Data	Read-only data, strings
Text	Program, machine code



- Instruction pointer is address in text segment
- Stack pointer is lower-most address of stack segment
- Program break pointer (BRK) is upper-most address of heap segment

Typical Process Address Space Layout

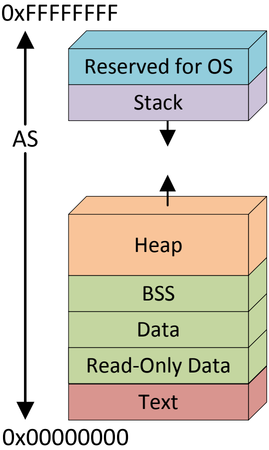
OS	Addresses where the kernel is mapped (cannot be accessed by process)
Stack	Local variables, function call parameters, return addresses
Heap	Dynamically allocated data (`malloc`)
BSS	Uninitialized local variables declared as static
Data	Initialized data, global variables
RO-Data	Read-only data, strings
Text	Program, machine code



- Instruction pointer is address in text segment
- Stack pointer is lower-most address of stack segment
- Program break pointer (BRK) is upper-most address of heap segment

Typical Process Address Space Layout

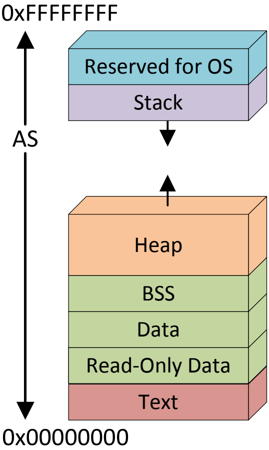
OS	Addresses where the kernel is mapped (cannot be accessed by process)
Stack	Local variables, function call parameters, return addresses
Heap	Dynamically allocated data (`malloc`)
BSS	Uninitialized local variables declared as static
Data	Initialized data, global variables
RO-Data	Read-only data, strings
Text	Program, machine code



- Instruction pointer is address in text segment
- Stack pointer is lower-most address of stack segment
- Program break pointer (BRK) is upper-most address of heap segment

Typical Process Address Space Layout

OS	Addresses where the kernel is mapped (cannot be accessed by process)
Stack	Local variables, function call parameters, return addresses
Heap	Dynamically allocated data (`malloc`)
BSS	Uninitialized local variables declared as static
Data	Initialized data, global variables
RO-Data	Read-only data, strings
Text	Program, machine code



- Instruction pointer is address in text segment
- Stack pointer is lower-most address of stack segment
- Program break pointer (BRK) is upper-most address of heap segment

Compiling, Linking, and Loading Programs

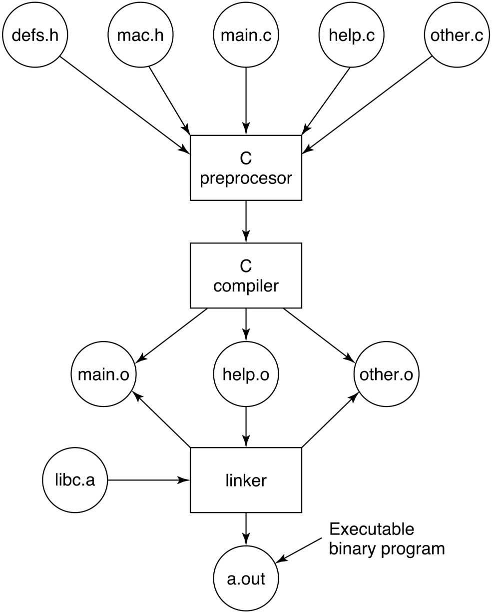
The C Programming Language

- 1966 Martin Richards creates BCPL for building compilers and OS
  - AmigaOS was originally written in BCPL
- 1969 Ken Thompson creates B, a simpler version of BCPL for PDP-7
- 1969-1973 Dennis Ritchie develops C for PDP-11
  - Highly influenced by B
  - Maps efficiently to machine instructions → well suited for OS development
  - Origin closely tied to the development of the first Unix
    - Unix originally written in assembly for PDP-7, then ported to C for PDP-11
- C deals with the same objects that computers do
  - Numbers, characters (basic data types), and addresses (pointers)
  - Compositions of the above (arrays, structures)
  - Calls, jumps, and conditional branches (Functions, loops, if/switch)

Read the "K&R", every computer scientist should read it:
Kernighan and Ritchie's "The C Programming Language"

The C Build Process

- Headers (.h) are `#included` and macros are expanded in the preprocessor before passing a `.c` file to the compiler
- There, each `.c` file is compiled to an object (`.o`) file
- The Linker combines all `.o` files to an executable binary
  - The `.o` files need exactly one main function, the starting point of the program
  - Static libraries (`.a`) may also be passed
- Run `nm`, `objdump`, and `readelf` on `.o` and `a.out` files!



Compiling – Object Code Files

- The object contains instructions and data generated by the compiler
- Objects are structured by
  - Segments (sections), e.g., text segment, data segment
  - Labels, i.e., function names (not associated with virtual addresses, yet)

square.h

```
#ifndef SQUARE_H_
#define SQUARE_H_
int square( int );
#endif
```

square.c

```
#include "square.h"
int square( int a )
{
    return a * a;
}
```

gcc -m32 -fno-builtin -c square.c

objdump -d square.o

square.o

```
Disassembly of section .text:

00000000 <square>:
0:    55                    push    %ebp
1:    89 e5                 mov     %esp,%ebp
3:    8b 45 08              mov     0x8(%ebp),%eax
6:    0f af 45 08           imul    0x8(%ebp),%eax
a:    5d                    pop     %ebp
b:    c3                    ret
```

Linker

- The Linker (Linux: `ld`) builds the executable from object files by
  - Arranging segments in non-overlapping memory regions
  - Constructing a global symbol table which maps labels to addresses
  - Patching addresses in code (relocation)
  - Writing the result to the binary file
- C++ can contain multiple functions with the same name (overloading)
  - Compiler mangles function name, parameters (, and compiler version) to obtain label → Mangling not compatible across compiler versions!
- In C, the label name is the function name
  - Linker cannot build binary if multiple functions with the same name exist

Dynamic Linking

- May want to load plugins (e.g., kernel modules/drivers) at runtime!
  - Don’t have to link everything before writing executable
  - Linking is nothing magical → can link at runtime!

dyn_square.c

```
int dyn_square( int a )
{
   return a * a;
}
```

main.c

```
void *dyn = dlopen( "dyn_square.o", RTLD_LAZY );
int (*square)(int) = dlsym( dyn, "dyn_square" );
square( 42 );
```

Loader – Executing a Program

- When starting a program, the loader
  - Reads code/data segments from disk into buffer cache
  - Maps code read-only and executable
  - Initializes rw-data and data sections (maps them accordingly)
  - Allocates space for the heap (`sbrk`), stack (e.g., 8 MiB)
  - Allocates space for BSS and nulls it

- In reality: Lots of optimizations
  - Code/data already in cache? Don't read from disk again
  - Stack space allocated when used
  - BSS not allocated and initialized until used
  - Code lazily loaded when it is first used (demand loading)
  - Share code with already running processes

Static Shared Libraries

- "Everyone" links standard library
  - Need to have copies of standard library functions in every executable

- Goal: Have shared library file that can be used by "everyone"
- Idea: Static shared libraries
  - Define shared library segment in all processes that use that library
  - Linker links executable against library segment but doesn't copy it into binary file
  - Loader brings shared library into the buffer cache only once
    → shares section among all processes that use it (demand loading)

- Problem: Now all programs need to have library **at same place in virtual address space**!
  - What if another library already occupies that space?
  - Needs system-wide pre-allocation of address
  - What if sum of all libraries gets too big for address space?

Dynamic Shared Libraries

- Idea: Dynamic shared libraries
  - Allow loading shared library at any virtual address

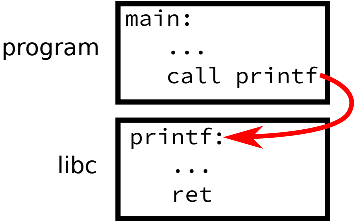
- New problems:
  - How do you call functions if the position varies?
  - Runtime linking would prevent code sharing!

  → "All problems in computer science can be solved by another level of **indirection**" (David Wheeler)

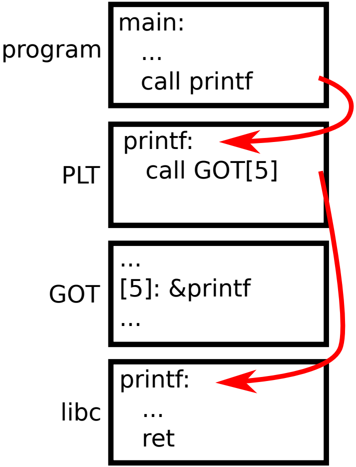
- Solutions: Position-independent code (PIC)
  - Procedure linkage table (PLT) Table that contains stubs pointing to the GOT for functions that are linked in dynamically
    → Now both PIC and non-PIC code can link!
  - Global offset table (GOT) Table that maps stubs to functions in various dynamic libraries
  - Lazy dynamic binding Link each function on its first call, not at startup

Static vs. Dynamic Shared Libraries

Static Shared Library



Dynamic Shared Library



Summary

- Processes
  - Program : Recipe is like Process : Cooking
  - Processes are a resource container for the OS
  - For the process it feels like it is alone: it has its own CPU and memory
  - The OS implements multiprogramming by rapidly switching processes
- Compiler: Creates an object file from each source file
  - Incomplete view of the world
  - Names functions and variables symbolically
- Linker: Combines object files to an executable
  - Resolves virtual addresses
  - Decides where everything lives
  - Finds and updates references to symbols (labels)
- Loader: Brings executable in memory and starts program

Further Reading

- Tanenbaum/Bos, “Modern Operating Systems”, 4th Edition: Pages 73, 85–97
- Drepper, “How to Write Shared Libraries” (aka “dso howto”)