Operating Systems

Introduction to Virtualizing Memory

Hongliang Liang, BUPT

Sep. 2022

OS: managing shared resources

Sharing resources over time: Physical processors

- The main topic of past lectures

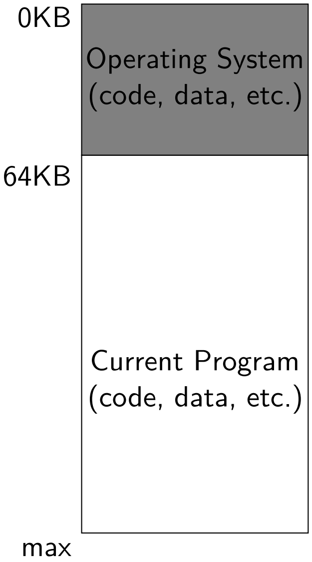
Sharing resources over space: Memory

- The main topic of upcoming lectures
- There is never enough memory
- "640 KB ought to be enough for anyone."

Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access **directly**
- Memory unit only sees a stream of addresses + read or write requests
- Register access in one CPU clock cycle (or less)
- Main memory acesses take many cycles, causing a **stall**
- **Cache** (on the CPU chip) sits between main memory and CPU registers
- Protection of memory is required to ensure correct operation

Memory in early operating systems



Multiprogramming and time sharing

We need to accommodate multiple processes now

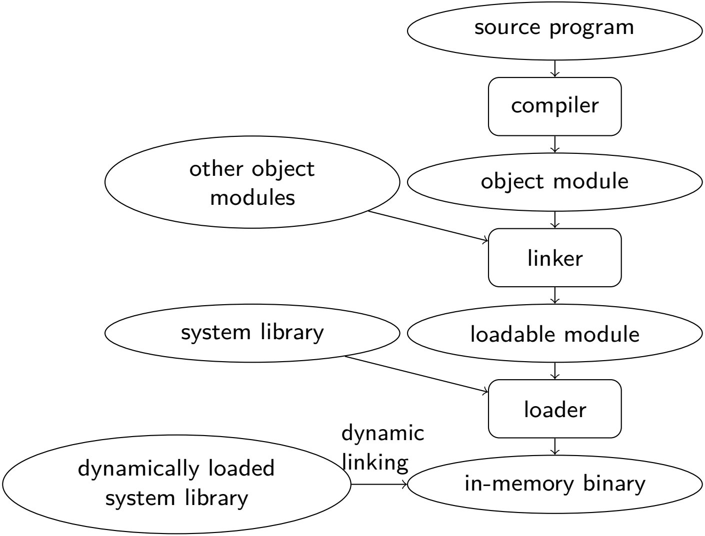
- Multiprogramming: multiple batch jobs run at the same time
- Time sharing: Multiple users using interactive processes at the same time

The important question is how?

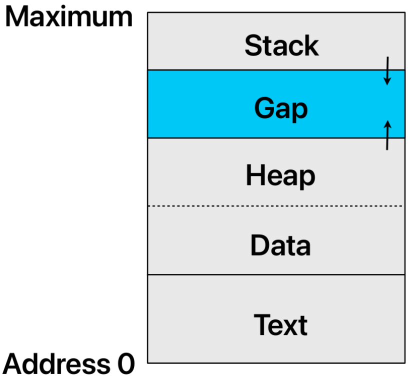
Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
- Compile time: If memory location known a priori, absolute code can be generated; must recompile code if starting location changes
- Load time: Must generate relocatable code if memory location is not known at compile time
- Execution time: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - Need hardware support for address maps (e.g., base and limit registers)

Multistep Processing of a User Program



Address space in a process: revisited



Why do we need dynamic memory?

Why do we need dynamic memory?

- The amount of required memory may be task dependent
- Input size may be unknown at compile time
- Conservative pre-allocation would be wasteful
- Recursive functions (invocation frames)

Excursion: procedure invocation frames

Calling a function allocates an invocation frame to store all local variables and the necessary context to return to the callee.

```
int called(int a, int b) {
  int tmp = a * b;
  return tmp / 42;
}
void main(int argc, char *argv[]) {
  int tmp = called(argc, argc);
}
```

What data is stored in the invocation frame of `called`?

Excursion: procedure invocation frames

Calling a function allocates an invocation frame to store all local variables and the necessary context to return to the callee.

```
int called(int a, int b) {
  int tmp = a * b;
  return tmp / 42;
}
void main(int argc, char *argv[]) {
  int tmp = called(argc, argc);
}
```

What data is stored in the invocation frame of `called`?

- Slot for `int tmp`
- Slots for the parameters a, b
- Slot for the return code pointer
- Order in most ABIs: b, a, RIP, tmp

Excursion: procedure invocation frames

Calling a function allocates an invocation frame to store all local variables and the necessary context to return to the callee.

```
int called(int a, int b) {
  int tmp = a * b;
  return tmp / 42;
}
void main(int argc, char *argv[]) {
  int tmp = called(argc, argc);
}
```

What data is stored in the invocation frame of `called`?

- Slot for `int tmp`
- Slots for the parameters a, b
- Slot for the return code pointer
- Order in most ABIs: b, a, RIP, tmp

The compiler creates the necessary code, according to the ABI.

Stack for procedure invocation frames

- The stack enables simple storage of function invocation frames
- Stores calling context and sequence of active parent frames
- Memory allocated in function prologue, freed when returned

Stack for procedure invocation frames

- The stack enables simple storage of function invocation frames
- Stores calling context and sequence of active parent frames
- Memory allocated in function prologue, freed when returned

What happens to the data when function returns?

Stack for procedure invocation frames

- The stack enables simple storage of function invocation frames
- Stores calling context and sequence of active parent frames
- Memory allocated in function prologue, freed when returned

What happens to the data when function returns?

- Data from previous function lingers, overwritten when the next function initializes its data

Quiz: scopes, stack, and persistence

```c
int a = 2;
int called(int b) {
  int c = a * b;
  printf("a: %d b: %d c: %d\n", a, b, c);
  a = 5;
  return c;
}
int main(int argc, char* argv) {
  int b = 2, c = 3;
  printf("a: %d b: %d c: %d\n", a, b, c);
  b = called(c);
  printf("a: %d b: %d c: %d\n", a, b, c);
  return 0;
}
```

Answer:

```
a: 2 b: 2 c: 3
a: 2 b: 3 c: 6
a: 5 b: 6 c: 3
```

Dynamic data structure: heap

A heap of randomly allocated memory objects with *statically unknown size* and *statically unknown allocation patterns*. The size and lifetime of each allocated object is unknown.

API: `alloc` creates an object, `free` indicates it is no longer used.

Dynamic data structure: heap

A heap of randomly allocated memory objects with *statically unknown size* and *statically unknown allocation patterns*. The size and lifetime of each allocated object is unknown.

API: `alloc` creates an object, `free` indicates it is no longer used.

How would you manage such a data structure?

Heap: straw man implementation

```c
char storage[4096], *heap = storage;
char *alloc(size_t len) {
  char *tmp = heap;
  heap = heap + len;
  return tmp;
}

void free(char *ptr) {}
```

- Advantage: simple
- Disadvantage: no reuse, will run out of memory

Heap: free list

Idea: abstract heap into list of free blocks.

- Keep track of free space, program handles allocated space
- Keep a list of all available memory objects and their size

Implementation:

- `alloc`: take a free block, split, put remainder back on free list
- `free`: add block to free list

Heap and OS interaction

- The OS hands the process a large chunk of memory to store heap objects
- A runtime library (the libc) manages this chunk
- Memory allocators aim for performance, reliability, or security

Quiz: where is it?

```c
int g;
int main(int argc, char *argv[]) {
  int foo;
  char *c = (char*)malloc(argc*sizeof(int));
  free(c);
}
```

Possible storage locations: stack, heap, globals, code

Quiz: where is it?

```c
int g;
int main(int argc, char *argv[]) {
  int foo;
  char *c = (char*)malloc(argc*sizeof(int));
  free(c);
}
```

Possible storage locations: stack, heap, globals, code

- Stack: argc, argv, foo, c
- Heap: *c
- Globals: g
- Code: main

Address space: an abstraction

- The user process uses **virtual addresses** in its own address space
- The virtual memory system in the OS is responsible for virtualizing physical memory and provides the abstraction of address spaces to user processes
- But how can the OS build this abstraction of a private, potentially large address space for multiple processes on top of a single, physical memory?

Goals of virtualizing memory

Before we introduce more ideas, let's first think about our goals

- Transparency
- Efficiency
- Protection

How to do?

Address Translation

Translating addresses at run time

Transforms each memory address (instruction fetch, load, store)

- From the virtual address provided by the instruction to its corresponding physical address
- This is to be performed at every memory reference since we need transparency
- But we also need efficiency!

Hardware Support: Memory Management Unit (MMU) — as part of the CPU

What does the MMU do?

Virtual memory addresses in an address space

Address translation performed on the fly during program execution

Memory Management Unit

Physical memory addresses in the physical memory

The OS has to get involved to set up the hardware

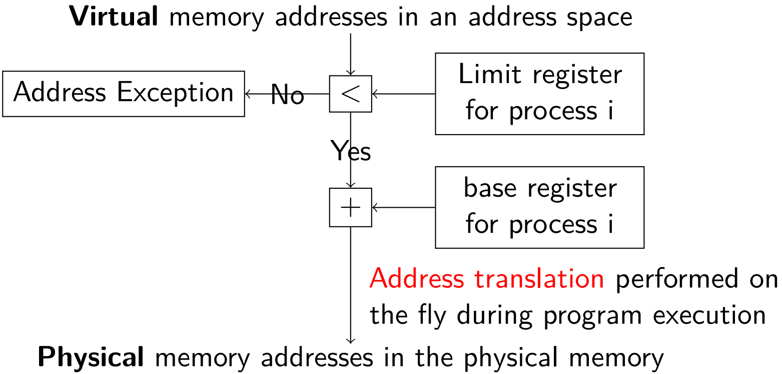
Three assumptions to get started

- A user process's address space must be placed **contiguously** in physical memory
- The size of the address space is **less than** the size of the physical memory
- Each address space is exactly the **same size**

Dynamic Relocation

- MMU has one **base** and one **bounds (or limit)** register
- Base register converts each virtual address to a physical address by adding an offset — relocation
- Bounds (limit) register keeps memory references within bounds — protection
- OS assigns each process a separate base and limit register value when a process is started

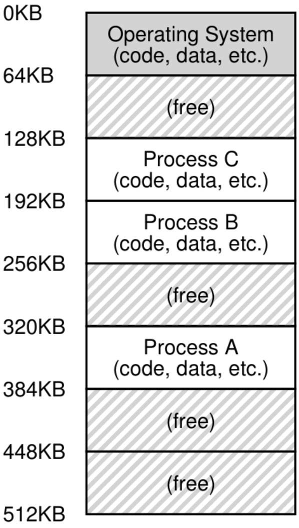
Dynamic relocation: Base and bounds registers



When a process is created, how can the OS find space in the physical memory for its new address space?

Given our assumptions: fixed size and less than physical memory

Simple idea: maintain a free list



The work that the OS must do

When a process is created

- Find a free entry in the free list and mark it as used

When a process is terminated (killed or exits gracefully)

- Returns its memory back to the free list

During a context switch

- Save and restore the base-and-bounds registers in the Process Control Block (PCB)

Any problems with base-and-bounds virtualization?