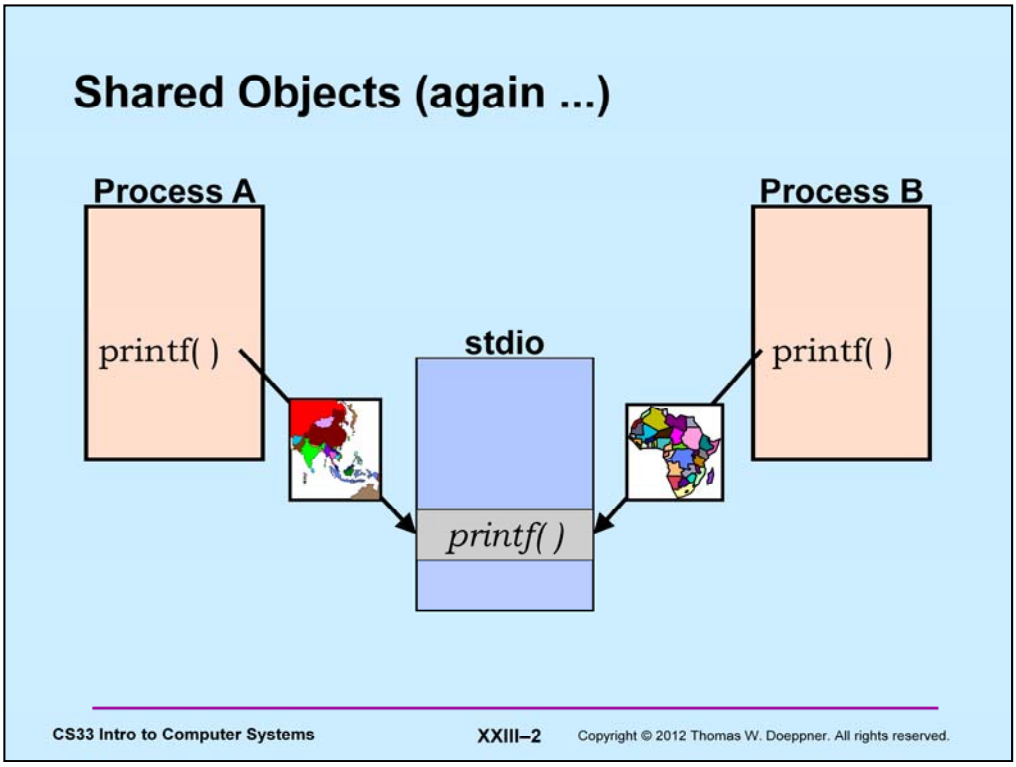
# CS 33

## Virtual Memory (2)

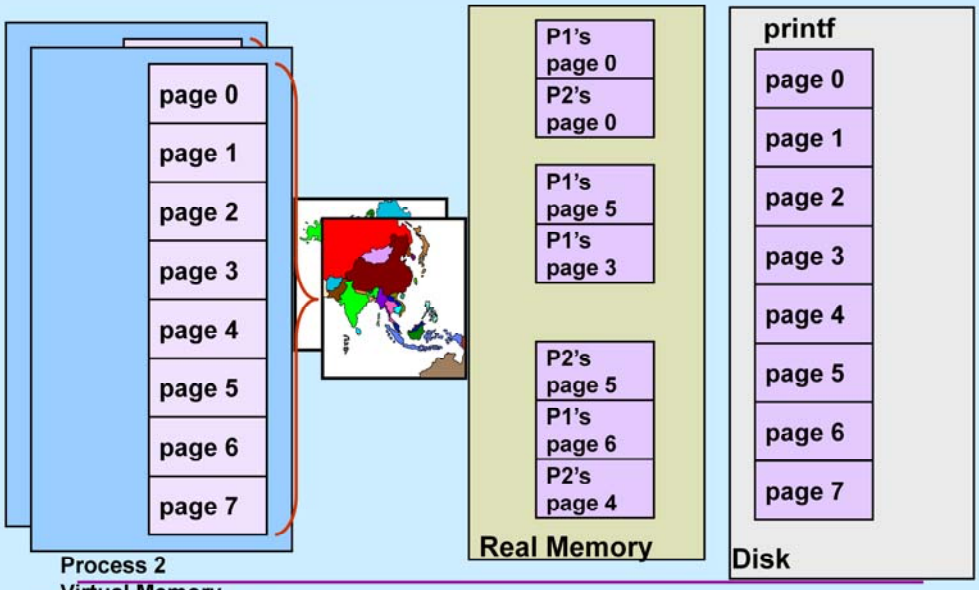


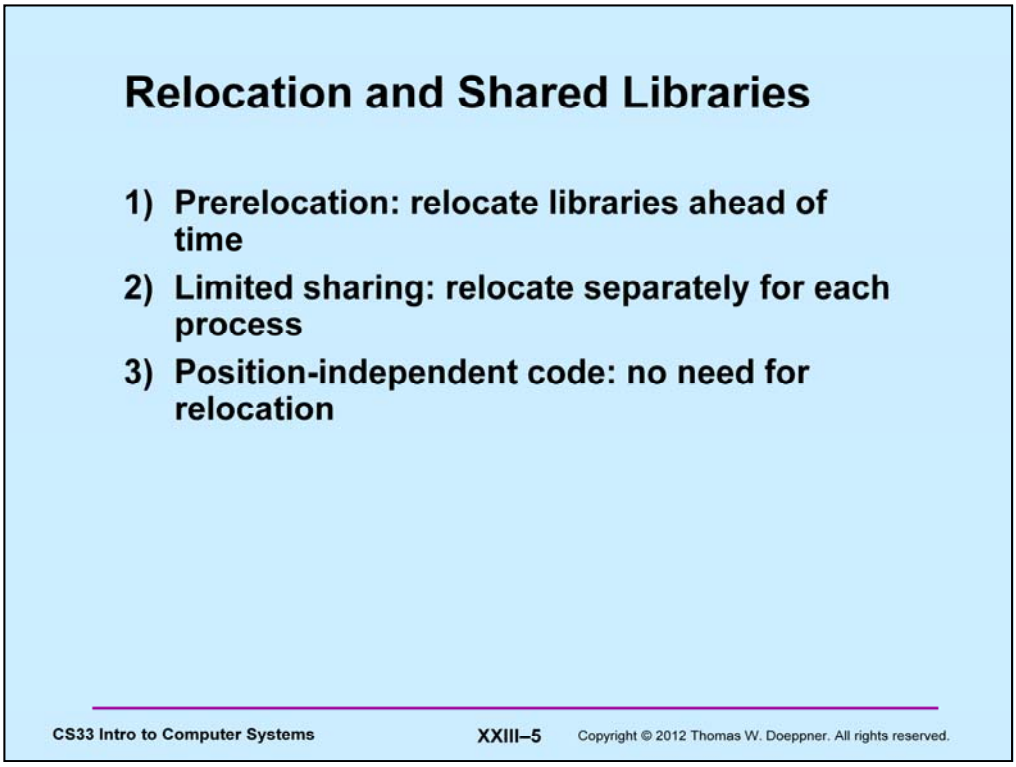

We've discussed the use of shared objects already. Now we'll delve into how they're actually made to work. The general idea is that we'd like to somehow map the shared object into the address spaces of processes that are using them.

# Mapping printf into the Address Space

- Printf's text
  - read-only
  - can it be shared?
- Printf's data
  - read-write
  - not shared with other processes
  - initial values come from file
- Can mmap be used?
  - MAP_SHARED wouldn't work
    - » changes made to data by one process would be seen by others
  - MAP_PRIVATE does work!
    - » mapped region is initialized from file
    - » changes are private

# Mapping printf

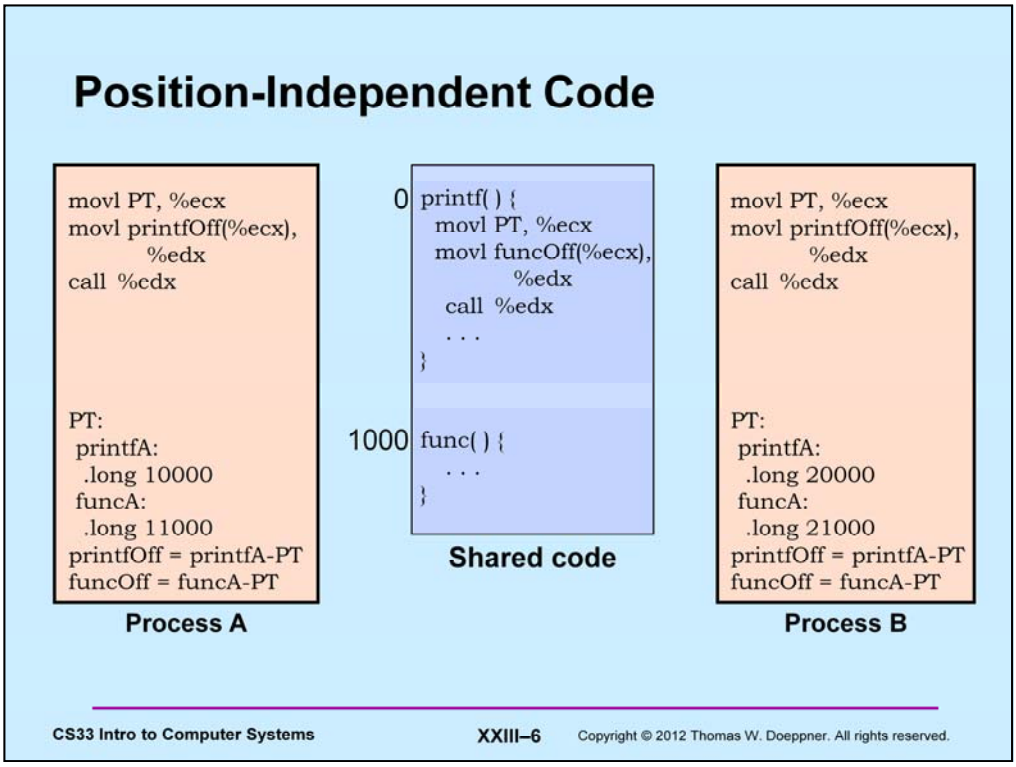




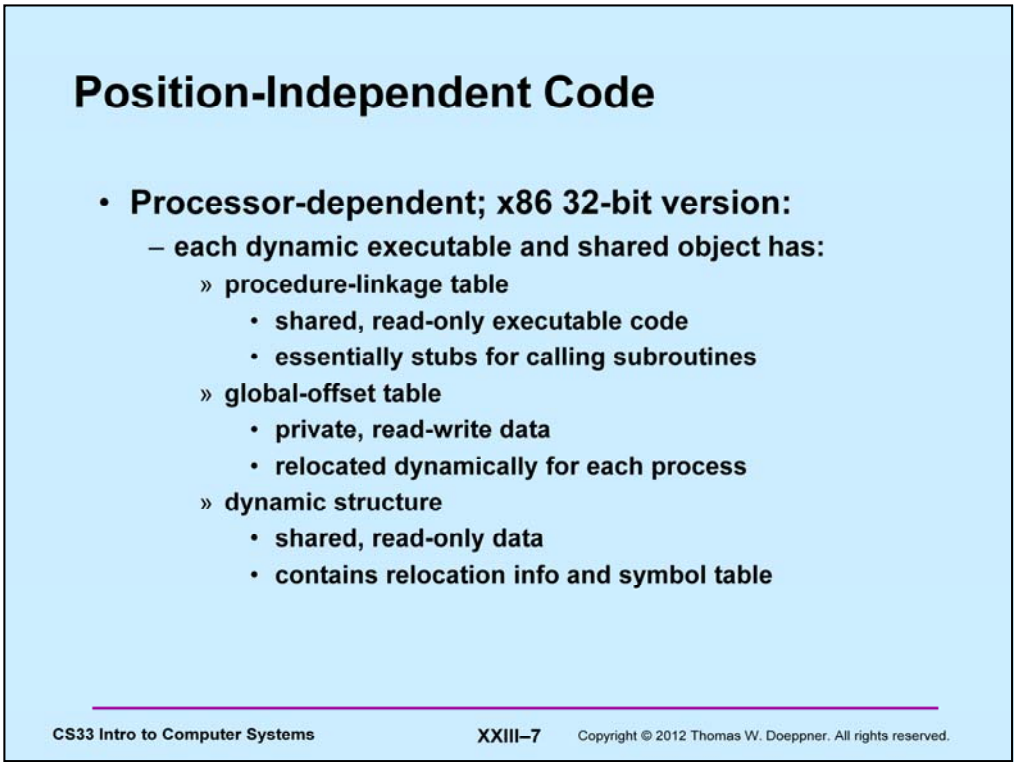

Clearly all processes using printf will have to have their own private copy of its data and bss areas, perhaps set up with the use of mmap with the MAP_PRIVATE option. But what about printf's text? If we didn't have to worry about relocation, then all processes could share it. If all users of *printf* agree to load it and everything it references into the same locations in their address spaces, we would have no relocation problem, and thus the text could be shared among all processes. But such agreement is, in general, hard to achieve. It is, however, the approach used in Windows.

A possibility might be for the processes using *printf* to mmap it into their address spaces with the MAP_PRIVATE option, then for each process to relocate its copy. This definitely works, but is a bit cumbersome.

Another possibility is for *printf* to be written in such a way that relocation is not necessary. Code written in this fashion is known as *position-independent code* (PIC).

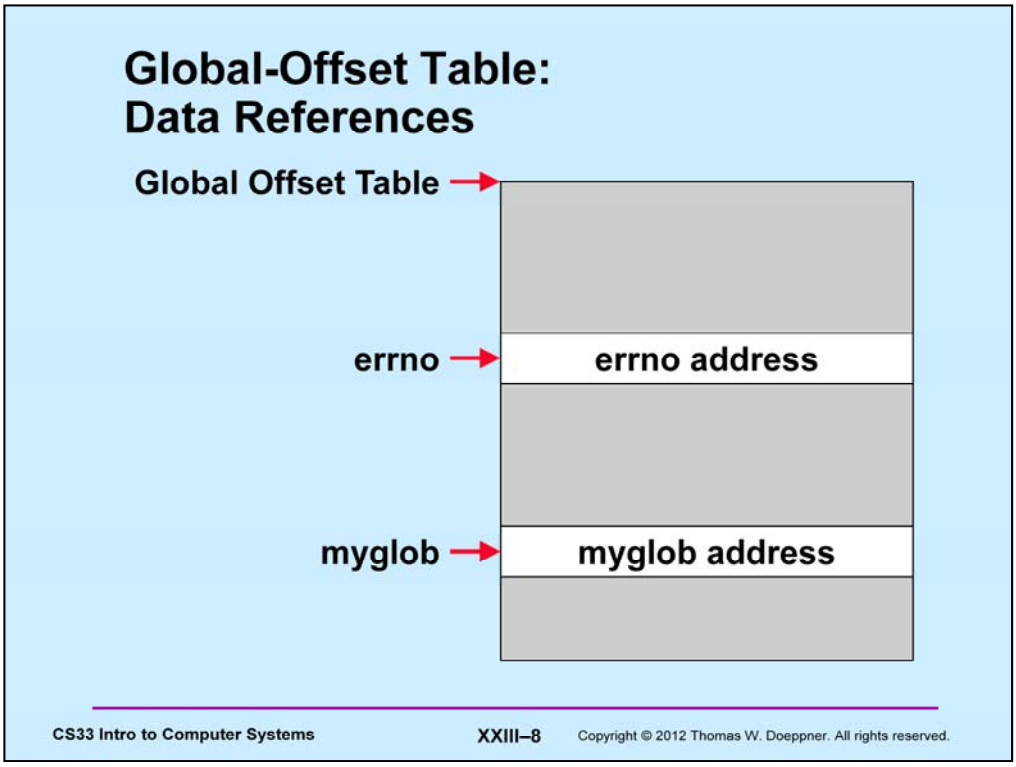



As we'll soon see, how procedures are called using position-independent (PIC) code on Linux is rather complicated, so here is a somewhat simplified example of how it might be done. Processes A and B are sharing a library containing *printf* (note that *printf* contains a call to another shared routine, *func*), though each has it mapped into a different location. Each process maintains a private table at location PT (PT's address is different in each process). In the table are the addresses of the shared routines, as mapped into the process (printf is at location 10,000 in process A and at location 20,000 in process B). Also defined is the offset (printfOff and funcOff) of each address relative to the start of the table. Thus, rather than call a routine directly (via an address embedded in the code), a position-independent call is made: the address of the table is loaded into register ecx. Then the table entry containing the desired address is loaded into register edx. Finally, a call is placed to the address contained in edx.




To provide position-independent code on a 32-bit x86, ELF requires three data structures for each dynamic executable (i.e., the program binary loaded by *exec*) and shared object: the *procedure-linkage table,* the *global-offset table*, and the *dynamic structure*. To simplify discussion, we refer to dynamic executables and shared objects as *modules*. The procedure linkage table contains the code that's actually called when control is to be transferred to an externally defined routine. It is shared by all processes using the associated executable or object, and makes use of data in the global-object table to link the caller to the called program. Each process has its own private copy of each global-object table. It contains the relocated addresses of all externally defined symbols. Finally, the dynamic structure contains much information about each module. What is used for linking is relocation information and the symbol table, as we explain in the next few slides.

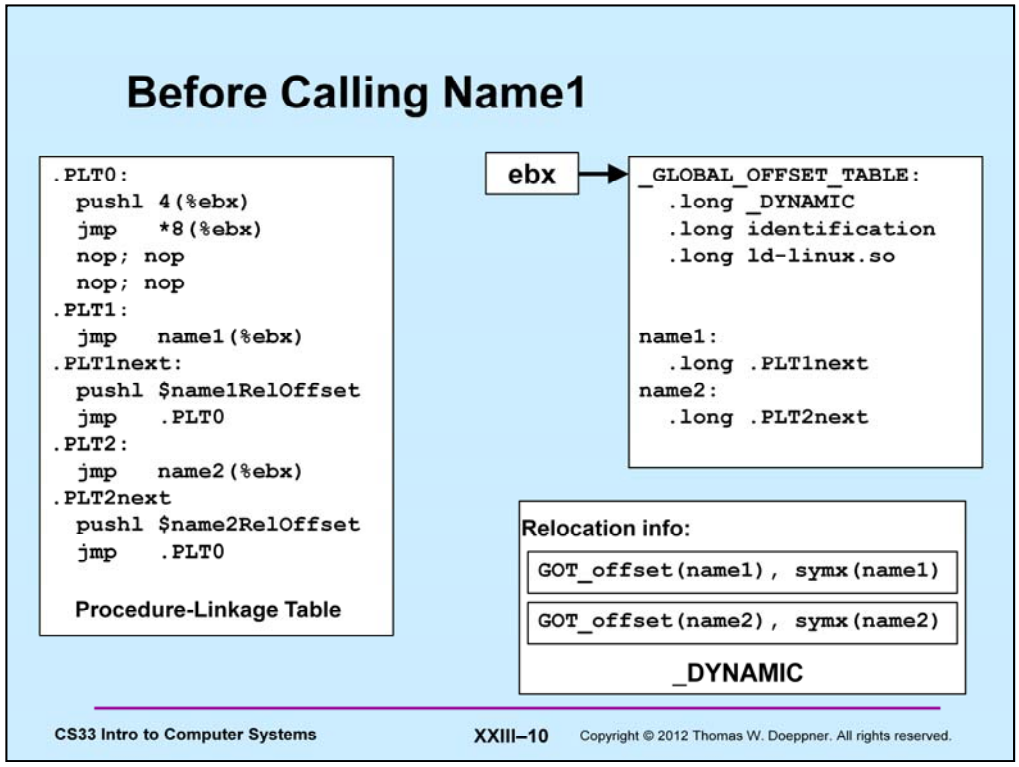
How things work is similar for other architectures, but definitely not the same.



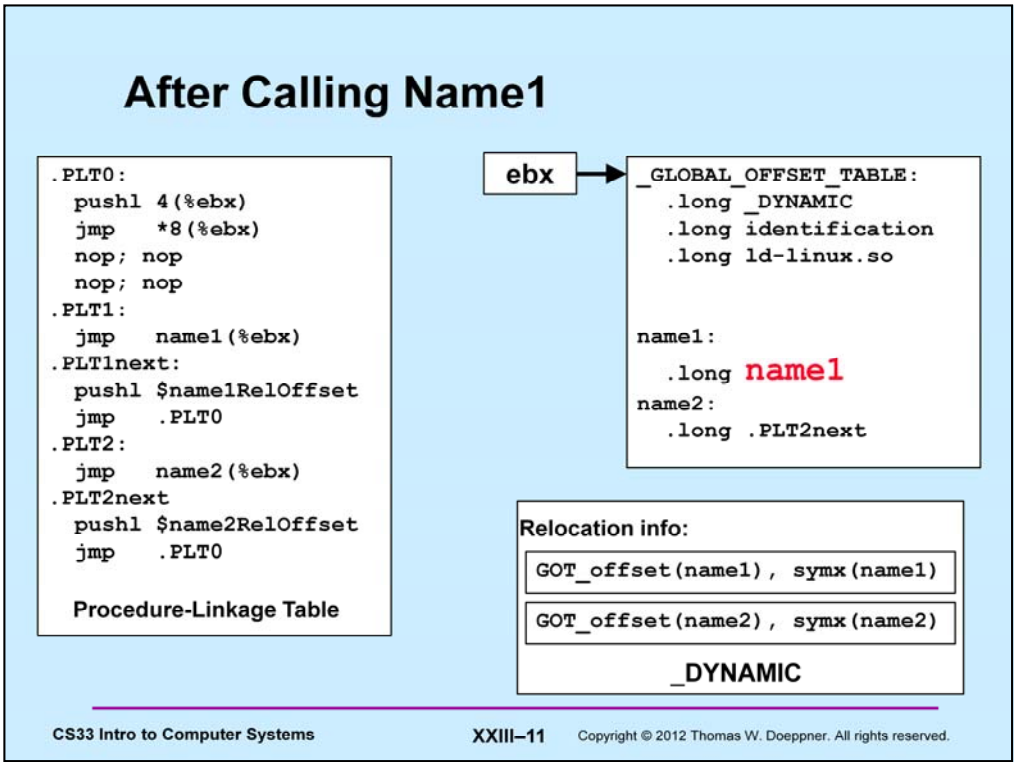

To establish position-independent references to global variables, the compiler produces, for each module, a *global-offset table*. Modules refer to global variables indirectly by looking up their addresses in the table. The way this works is that when code in the module refers to something in the table, a register is set to point to it. The item needed is at some fixed offset from the beginning of the table. When the module is loaded into memory, ld.so is responsible for putting into it the actual addresses of all the needed global variables.

# Procedure References

- Lots of them
- Many are never used
- Fix them up on demand




Dealing with references to external procedures is considerably more complicated than dealing with references to external data. This slide shows the procedure linkage table, global offset table, and relocation information for a module that contains references to external procedures *name1* and *name2*. Let's follow a call to procedure name1. The general idea is before the first call to name1, the actual address of the name1 procedure is not recorded in the global-offset table, Instead, the first call to name1 actually invokes ld-linux.so, which is passed parameters indicating what is really wanted. It then finds name1 and updates the global-offset table so that things are more direct on subsequent calls. To make this happen, references from the module to name1 are statically linked to entry .PLT1 in the procedure-linkage table. This entry contains an unconditional jump to the address contained in the name1 offset of the global-offset table (pointed to by register ebx). Initially this address is of the instruction following the jump instruction, which contains code that pushes onto the stack the offset of the name1 entry in the relocation table. The next instruction is an unconditional jump to the beginning of the procedure-linkage table, entry .PLT0. Here there's code that pushes onto the stack the second 32-bit word of the global-offset table, which contains a value identifying this module. The following instruction is an unconditional jump to the address in the third word of the global-offset table, which is conveniently the address of ld-linux.so. Thus control finally passes to ld-linux.so, which looks back on the stack and determines which module has called it and what that module really wants to call. It figures this out based on the module-identification word and the relocation table entry, which contains the offset of the name1 entry in the global-offset table (which is what must be updated) and the index of name1 in the symbol table (so it knows the name of what it must locate).




Finally, ld-linux.so writes the actual address of the name1 procedure into the name1 entry of the global-offset table, and, after unwinding the stack a bit, passes control to name1. On subsequent calls by the module to name1, since the global-offset table now contains name1's address, control goes to it more directly, without an invocation of ld-linux.so.