**CMPSCI 377 Operating Systems** **Fall 2012**

# Lecture 11: October 16

*Lecturer: Prashant Shenoy* *TA: Sean Barker & Demetre Lavigne*

## 11.1 Finishing Up Deadlocks: Banker's Algorithm

The Banker's algorithm is a resource allocation and deadlock avoidance algorithm developed by Edsger Dijkstra. It tests for safety by simulating the allocation of pre-determined maximum possible amounts of all resources, and then makes a "safe-state" check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue. The Banker's algorithm is run by the operating system whenever a process requests resources. The algorithm prevents deadlock by postponing the request if it determines that accepting the request could put the system in an unsafe state (one where deadlock could occur).

### 11.1.1 Resources

For the Banker's algorithm to work, it needs to know three things:

1. How much of each resource each process could possibly request
2. How much of each resource each process is currently holding
3. How much of each resource the system has available

### 11.1.2 Safe and Unsafe States

A state is considered safe if it is possible for all processes to finish executing (terminate) by following some execution sequence. The system assumes that all processes will eventually attempt to acquire their stated maximum resources and terminate. Given that assumption, the algorithm determines if a state is safe by trying to find a hypothetical set of requests by the processes that would allow each to acquire (one-by-one) its maximum resources and then terminate (returning its resources to the system). Any state where no such set exists is an unsafe state. In a safe state, at least one process should be able to acquire its maximum possible set of resources, and proceed to termination.

### 11.1.3 The simplified algorithm

When the system receives a request for resources, it runs the Banker's algorithm to determine if it is safe to grant the request. The algorithm is fairly straight forward once the distinction between safe and unsafe states is understood.

1. Can the request be granted?
    - If not, the request is impossible and must either be denied or put on a waiting list

2. Assume that the request is granted
3. Is the new state safe?
    - If so grant the request
    - If not, either deny the request or put it on a waiting list

Whether the system denies or postpones an impossible or unsafe request is a decision specific to the operating system.

Note: Look at lecture slides for the *pseudocode* and *examples*.

### 11.1.4 Trade-offs

Like most algorithms, the Banker's algorithm involves some trade-offs. Specifically, it needs to know how much of each resource a process could possibly request. In most systems, this information is unavailable, making the Banker's algorithm useless. Besides, it is unrealistic to assume that the number of processes is static. In most systems the number of processes varies dynamically. Moreover, the requirement that a process will eventually release all its resources (when the process terminates) is sufficient for the correctness of the algorithm, however it is not sufficient for a practical system. Waiting for hours (or even days) for resources to be released is usually not acceptable.

## 11.2 Memory Management

Memory management is the act of allocating, removing, and protecting computer memory for multiple processes. In its simpler forms, this involves providing ways to allocate portions of memory to programs at their request, and freeing it for reuse when no longer needed.

Initially every program executable is resident on disk. The OS loads the program from the disk into main memory; how memory is allocated and where that memory is reserved is determined by the memory manager. While executing the program, the CPU fetches instructions and data from memory, possibly requiring further interactions with the memory manager.

### 11.2.1 Terminology

- **Segment:** A chunk of memory assigned to a process.
- **Physical Address:** A physical address, also real address or binary address, is the actual physical memory address that is used to access a specific storage cell in main memory.
- **Virtual Address:** An address *relative* to the start of a process' address space. Also called the logical address. Virtual addresses are used because, typically, the program or compiler doesn't know where it will be located in memory.

### 11.2.2 Generation of addresses

There are several techniques that can be used to determine how addresses are generated for use by a program.

**Compile time:** The compiler generates the exact physical location in memory starting from some fixed starting position k. The OS is not involved here. This is very restrictive because the compiler must know ahead of time how all memory in the system is going to be allocated in order to prevent using an address that might be used by another application.

**Load time:** Compiler generates an address, but at load time the OS determines the process' starting position. Once the process loads, it does not move in memory. This allows for greater flexibility, but still restricts the process location once it has been started.

**Execution time:** Compiler generates an address, but the OS can place it anywhere in memory. This is the most flexible technique because the OS can remap how the compiled addresses relate to physical memory addresses on the fly.

## 11.3 Uniprogramming

Perhaps the simplest model for using memory is to provide uniprogramming without memory protection, where each application runs with a hardwired range of physical memory addresses. A uniprogramming environment allows only one application to run at a time, thus an application can use the same physical addresses every time, even across reboots. This means that there isn't much to do for memory management. However, only supporting a single process at a time prevents concurrency, and can reduce performance since multiple processes cannot be used to overlap periods of computation and I/O. Typically, uniprogramming applications use the lower memory addresses (low memory), and an operating system uses the higher memory addresses (high memory). In the simplest case, an application can address any physical memory location. More advanced systems protect the OS by checking all user program memory accesses against the OS memory bounds.

## 11.4 Multiprogramming

Multiprogramming operating systems support multiple applications at once. Ideally, the OS should do this *transparently*–applications should be unaware that memory is shared and they should not care where in physical memory they are allocated. Secondly, the OS must provide *safety*, to ensure that that processes cannot corrupt each other or the OS. Finally, the main goal of multiprogramming is to improve *efficiency*. Multiprogramming should not degrade performance badly due to the fact that more advanced memory management is required.

## 11.5 Relocation

Relocation is the simplest form of multiprogramming and it is the ability to execute processes independently from their physical location in memory. The role of relocation is central for memory management: virtually all the techniques in this field rely on the ability to relocate processes efficiently. The need for relocation is immediately evident when one considers that in a general-purpose multiprogramming environment a program cannot know in advance (before execution, i.e. at compile time) what processes will be running in memory when it is executed, nor how much memory the system has available for it, nor where it will be located. Hence a program must be compiled and linked in such a way that it can later be loaded starting from an unpredictable address in memory, an address that can even change during the execution of the process itself, if any swapping occurs.

It's easy to identify the basic requirement for a (binary executable) program to be relocatable: all the references to memory it makes during its execution must not contain absolute (i.e. physical) addresses of memory cells, but must be generated relatively, i.e. as a distance measured in contiguous memory words from some known point such as the start of the program's memory region. To support this, the *base* and *limit* addresses are used. These addresses refer to the first and last address of physical memory that a program can access respectively. Thus to ensure safety, all addresses generated for a process must reside within the base and limit addresses. Accessing an address outside this range can lead to what is called a "segmentation fault".

Relocation can be done in one of two ways. With *static* relocation, all addresses are generated once at load time. Once the program is running, it cannot be moved because there is no mechanism for recomputing the addresses that were previously determined. A *dynamic relocation* system generates all addresses during execution. In this case, the assembly code is produced with relative (or "logical") addresses for all data and instructions. These relative addresses are then added to the base address described previously. Typically, this is done in hardware using a special base (or "relocation") register so addresses can be calculated very quickly. To ensure protection, the address is also compared against the limit register to ensure that it is within the program's address space.

The benefit of dynamic relocation is that processes can be easily moved or grown during execution. This can be necessary if a process continuously allocates memory, or if a process must be moved in order to prevent memory fragmentation. However, there is some extra overhead since an addition is required before every memory access. In addition, the protection provided by base and limit registers can be overly restrictive, preventing sharing of memory between processes. Finally, the approach as described thus far, requires all processes to fit in memory, and limits the total size of a process to the available memory in the system. The dynamic relocation approach provides transparency and safety, and is reasonably efficient, but it has some limitations such as requiring full processes to be moved if they grow too large, which can be slow. The approaches so far also rely on contiguous regions of memory for each process.