Chapter 5 Process Synchronization

5.5 Mutex Locks

Operating systems designers build software tools to solve the critical-section problem. The simplest and most basic of these is the *mutex lock*.

```
do {
    acquire(&lock);
        /* critical section */
    release(&lock);
        /* remainder section */
} while (true);
```

A process must `acquire()` the lock before entering a critical section, and then `release()` it at after exiting the section. A process that attempts to acquire an unavailable lock is *blocked* until the lock is released. Thus the mutex lock protect critical regions and prevent race conditions. These operations are defined as:

```
acquire() {                  release() {
  while(!available)              available = true;
        ; /* busy wait */    }
    available = false;
}
```

Calls to either `acquire()` or `release()` must be atomic (usually implemented through `test_and_set()` and `compare_and_swap()`).

The main disadvantage of the implementation is that it requires busy waiting: waiting blocked in a loop for the call to `acquire()`, that is, the CPU is ”wasting” work by waiting in the loop. Busy waiting mutexes are also called *spinlock*. Spinlocks have an advantage: they require context switch (which are expensive) when locking a process (compare it to condition variables). Thus, when locks are expected to be held for only short times, spinlocks may be cheaper than other kinds of locks.

5.6 Semaphores

A *semaphore S* is an integer variable accessed with atomic operations `wait()` and `signal()`, defined as follows:

```
wait(S) {                      signal(S) {
    while(S <= 0)                  S++;
        ; /* busy wait */      }
    S--;
}
```

Kinds of Semaphores A *counting semaphore* can range over an unrestricted domain. A *binary semaphore* can range between 0 and 1 (thus are equivalent to mutex locks).

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. Init the semaphore to the total available resoureces. When a process wishes to use a resource, performs a `wait(S)` (decrementing the count of available resources). At resource release, perform `signal()` (incrementing the count).

Semaphore Implementation Rather than busy waiting (as in mutex locks), semaphores can be made to block the process when calling a `wait()` operation. Waiting processes are sent to a queue of waiting processes (waiting, in the sense of the operating system process).

When another process calls `signal()`, then a process in the list is restarted by a `wakeup()` operation, which changes the process from waiting state to ready state. A simple implemetation to this definition:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

Tip: One way to add and remove processes from the list so as to ensure bounded waiting is to use a FIFO queue,

5.7 Monitors

Several complex and tricky errors can ocur when programming with semaphores and locks. To deal with such errors, high-level language constructs called *monitors* are developed.

A *monitor* is an *abstract data type* (ADT) that includes a set of programmer defined operations that are provided with mutual exclusion within the monitor, thus ensuring that only one process at a time is active within the monitor.

Chapter 8 Main Memory

8.1 Background

Memory consists of a large array of bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter.

The memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data). Accordingly, we can ignore how a program generates a memory address. We are interested only in the sequence of memory addresses generated by the running program.

8.1.1 Basic Hardware.

Main memory and the registers built into the processor itself are the only general-purpose storage that the CPU can access directly. Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices. If the data are not in memory, they must be moved there before the CPU can operate on them.

For proper system operation we must protect the operating system from access by user processes, and processes from accessing other processes' memory. Since the operating system shouldn't intervene between the CPU and its memory accesses (because of the resulting performance penalty), this protection is implemented at hardware level.

First we need to make sure that each process has a separate memory space. We can provide this protection by using two registers: a *base register*, which holds the smallest legal physical memory addres of the process; and the *limit register*, which specifies the size of the range of memory available for the process (so that all memory addresses `n` that satisfy `base <= n < base + limit`, are available for a given process with such base and limit registers).

Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers. Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error. The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction. Since privileged instructions can be executed only in kernel mode, and since only the operating system executes in kernel mode, only the operating system can load the base and limit registers.

8.1.2 Address Binding.

A program residing on disk as a binary executable file, to be executed, must be brought into memory and placed within a process. Depending on the memory management in use, the process may be moved between disk and memory during its execution. The processes on the disk that are waiting to be brought into memory for execution form the *input queue.*

Most systems allow a user process to reside in any part of the physical memory. Thus, although the address space of the computer may start at `0x00000`, the first address of the user process need not be `0x00000`.

A user program goes through several steps before being executed. Addresses may be represented in different ways during these steps.

Addresses in the source program are generally symbolic. A compiler typically binds these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning of this module"). The linkage editor or loader in turn *binds* the relocatable addresses to absolute addresses (such as `0x74014`). Each binding is a mapping from one address space to another.

Binding of instructions and data to memory addresses can be done at any step along the way:

- **Compile time.** If you know at compile time where the process will reside in memory, then *absolute code* can be generated. If at some time later, the starting locationchanges, it is necessary to recompile the code.

- **Load time.** If it is not known at compile time where the process will reside in memory, then the compiler must generate *relocatable code.* In this case, final binding is delayed until load time (of the program into memory).

- **Execution time.** If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Most general-purpose operating systems use this method.

8.1.3 Logical and Physical Address Space.

An address generated by the CPU is referred to as a *logical address*, whereas an address seen by the memory unit (the address loaded into the memory-address register of the memory) is commonly referred to as a *physical address.* The compile-time and load-time address-binding methods generate identical logical and physical addresses. The execution-time binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a *virtual address.* The set of all logical addresses generated by a program is a *logical address space.* The set of all physical addresses corresponding to these logical addresses is a *physical address space.* The run-time mapping from virtual to physical addresses is done by a hardware device called the *memory-management unit* (MMU). The user program deals with logical addresses and never deals with real physical addresses. The memory-mapping hardware converts logical addresses into physical addresses. The user program generates only virtual addresses and thinks that process runs in locations in the virtual address space, however these are maped to physical addresses before they are used.

8.1.4 Dynamic Loading

So far, it has been necessary for the entire program and all data of a process to be in physical memory for the process to execute. The size of a process has thus been limited to the size of physical memory. To obtain better memory-space utilization, we can use dynamic loading: a routine is not loaded until it is called.

8.1.5 Dynamic Linking and Shared Libraries

Dynamically linked libraries are system libraries that are linked to user programs when the programs are run. Dynamic linking, is similar to dynamic loading. Here, though, linking, rather than loading, is postponed until execution time.

This feature is usually used with system libraries, such as language subroutine libraries. Without this facility, each program on a system must include a copy of its language library in the executable image. This requirement wastes both disk space and main memory.

With dynamic linking, a *stub* is included in the image for each library- routine reference. The stub is a small piece of code that indicates how to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present. When the stub is executed, it checks to see whether the needed routine is already in memory. If it is not, the program loads the routine into memory.

8.2 Swapping

A process must be in memory to be executed. A process, however, can be *swapped* temporarily out of memory to a backing store and then brought back into memory for continued execution. Swapping makes possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system.

8.2.1 Standard Swapping

It involves moving processes between main memory and a backing store. The backing store is commonly a fast disk, which mus be large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images. The system maintains a *ready queue* of all proesses whose memory images are on the backing store and ready to run. Whenever the CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory. If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process. It then reloads registers and transfers control to the selected process. Context switch time is given by: $\Delta t = 2 \cdot size_{process}/r$, where r is transfer rate of disk and 2 comes from swap in swap out.

Clearly, it would be useful to know exactly how much memory a user process is using, not simply how much it might be using. Then we would need to swap only what is actually used, reducing swap time. For this method to be effective, the user must keep the system informed of any changes in memory requirements. Thus, a process with dynamic memory requirements will need to issue system calls (some `request_memory()` and `release_memory()`) to inform the operating system of its changing memory needs.

If we want to swap a process, we must be sure that it is completely idle. Of particular concern is any pending I/O.

Standard swapping is not used in modern operating systems. It requires too much swapping time and provides too little execution time. Modified versions of swapping, however, are found on many systems.

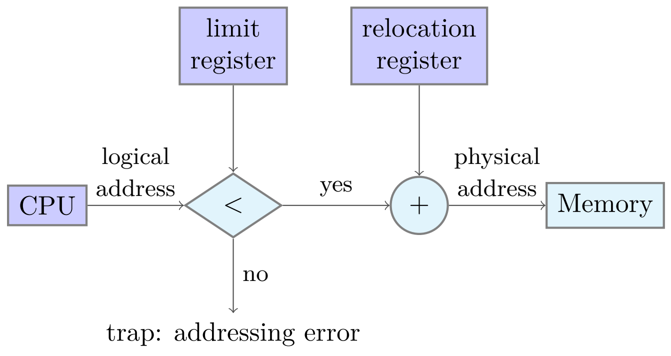
8.3 Contiguous Memory Allocation

The main memory must acommodate both the OS and various user processes. Contiguous memory allocation is an early solution to allocate memory in a somewhat efficient manner. In *contiguous memory allocation*, each process is contained in a single section of memory that is contiguous to the section containing the next process.

8.3.1 Memory Protection

The goal of memory protection is to prevent a process from accessing a region of memory it does not own. We can provide a simple solution with a syste having a *relocation register* (which contains smallest memory address reachable by the process) and a *limit register* (range of logical adresses of the process).

When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by a CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.



The relocation-register scheme provides an effective way to allow the operating system's size to change dynamically. Dynamically loaded operating system code is called *transient* os code.

8.3.2 Memory Allocation

One of the simplest methods for allocating memory is to divide memory into several fixed-sized *partitions*. Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the <u>number of partitions</u>. In this *multiple-partition* method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When process terminates, the partition becomes available for another process.

In the *variable-partition* scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered one large block of available memory, a hole. Eventually, as you will see, memory contains a set of holes of various sizes.

the memory blocks available comprise a *set* of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.

This procedure is a particular instance of the general *dynamic storage-allocation* problem, which concerns how to satisfy a request of size n from a list of free holes. There are many solutions to this problem:

- **First fit.** Allocate the first hole that is big enough.
- **Best fit.** Allocate the smallest hole that is big enough.
- **Worst fit.** Allocate the largest hole.

(Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization)

8.3.3 Fragmentation

External Fragmentation. Both the first-fit and best-fit strategies for memory allocation suffer from *external fragmentation.* As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous.

One solution to the problem of external fragmentation is compaction. The goal is to shuffle the memory contents so as to place all free memory together in one large block. Compaction is not always possible, however. If relocation is static and is done at assembly or load time, compaction cannot be done.

Another possible solution to the external-fragmentation problem is to permit the logical address space of the processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever such memory is available. Two complementary techniques achieve this solution: segmentation (Section 8.4) and paging (Section 8.5). These techniques can also be combined.

Fragmentation is a general problem in computing that can occur wherever we must manage blocks of data.

Internal Fragmentation. Memory fragmentation can be internal as well as external. Consider a multiple-partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes. The overhead to keep track of this hole will be substantially larger than the hole itself. The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size. With this approach, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is *internal fragmentation* —unused memory that is internal to a partition.

8.4 Segmentation

As a programmer, we think of memory as a collection of variable-sized segments with no necessary order among the segments. When writing a program, a programmer thinks of it as a main program with a set of methods, procedures, or functions, which may include various data strucures; without caring what adresses in memory these elements occupy. Segments vary in length, and the length of each is intrinsically defined by its purpose in the program. Elements within a segment are identified by their offset from the beginning of the segment (the first statement of the program, the seventh stack frame entry in the stack, the fifth instruction of the `Sqrt()`, ...).

Segmentation is a memory-management scheme that supports this programmer view of memory. A logical address space is a collection of segments.

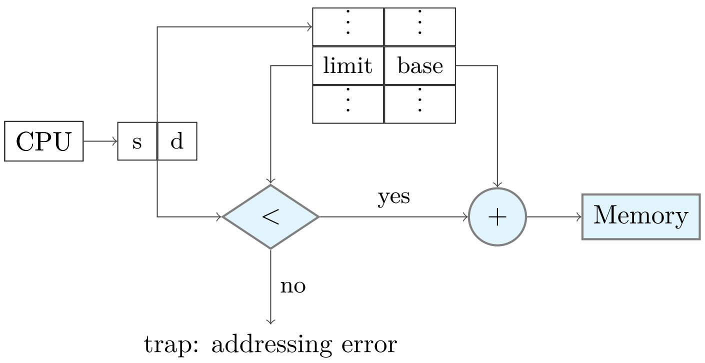
Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The programmer therefore specifies each address by two quantities: a segment name and an offset. For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name. Thus, a logical address consists of a two tuple:

$$(\text{segment-number}, \text{offset}) \equiv (s, d).$$

Normally, when a program is compiled, the compiler automatically constructs segments reflecting the input program.

8.4.1 Segmentation Hardware

We must define an implementaiton to map the two-dimensional user defined addresses (s, d) into the one-dimensional physical memory:



This mapping is effected by a *segment table.* Each entry in the segment table has a *segment base* and a *segment limit.* The segment base contains the starting physical address where the segment resides in memory, and the segment limit specifies the length of the segment.