Process Coordination & Semaphores

CISC 3320 — CUNY Brooklyn College
Lecture Notes

Process Coordination & Semaphores

In this chapter, we will:

1. Present a problem that happens when two processes attempt to write to the same shared memory location.
2. Explain why this problem happens and how to eliminate this problem.
3. Introduce Peterson's Solution, mutex (mutual exclusion), semaphores, and monitors.

Process Coordination & Semaphores

This video introduces the **critical region** problem and presents a software solution to it:

https://www.youtube.com/watch?v=eKKc0d7kzww

Process Coordination & Semaphores

Background: consider the following **Producer** and **Consumer** programs, which share access to the `buffer` array:

```
item nextProduced;

while( true ) {
    /* Produce an item and store it in nextProduced */
    nextProduced = makeNewItem( . . . );

    /* Wait for space to become available */
    while( ( ( in + 1 ) % BUFFER_SIZE ) == out )
        ; /* Do nothing */

    /* And then store the item and repeat the loop. */
    buffer[ in ] = nextProduced;
    in = ( in + 1 ) % BUFFER_SIZE;
}
```

```
item nextConsumed;

while( true ) {
    /* Wait for an item to become available */
    while( in == out )
        ; /* Do nothing */

    /* Get the next available item */
    nextConsumed = buffer[ out ];
    out = ( out + 1 ) % BUFFER_SIZE;

    /* Consume the item in nextConsumed
        ( Do something with it ) */
}
```

Process Coordination & Semaphores

The code we saw above is a pseudocode that contained only a part of each program.

The Producer program creates some item (number, string, or some other object) and places it inside the `buffer` array.

The Consumer program then reads the `buffer`'s items, and processes them, one by one.

These programs will work without any issues. The only concern is that the maximum number of items which can be placed into the buffer is `BUFFER_SIZE - 1`. One slot is unavailable because there always has to be a gap between the producer and the consumer.

We could try to overcome this deficiency by changing the programs by introducing a **`counter`** variable, as shown in the following slide. The `counter` variable will also be shared by both programs (both the Producer and Consumer can access and change `counter`.)

Process Coordination & Semaphores

These are the modified Producer and Consumer programs, which now use the `counter` variable (boldfaced below):

```
item nextProduced;

while( true ) {
    /* Produce an item and store it in nextProduced */
    nextProduced = makeNewItem( . . . );

    /* Wait for space to become available */
    while( counter == BUFFER_SIZE ) ; /* Do nothing */

    /* And then store the item and repeat the loop. */
    buffer[ in ] = nextProduced;
    in = ( in + 1 ) % BUFFER_SIZE;
    counter++;
}
```

```
item nextConsumed;

while( true ) {
    /* Wait for an item to become available */
    while( counter == 0 ) ; /* Do nothing */

    /* Get the next available item */
    nextConsumed = buffer[ out ];
    out = ( out + 1 ) % BUFFER_SIZE;
    counter--;

    /* Consume the item in nextConsumed
         ( Do something with it ) */
}
```

Process Coordination & Semaphores

After we made this change, the programs now suffer from a new issue that isn't noticed immediately called a race condition.

A **race condition** happens when two programs try modifying the same variable simultaneously, which could lead to uncertainty in which value the variable will contain (either the value that the 1st program assigned, the value that the 2nd program assigned, or some other value.)

Specifically to our programs, a race condition will occur with the change of the `counter` variable, at the statements: `counter++;` of the Producer program and `counter--;` of the Consumer program.

Example: If `counter` is currently `5`, and both programs execute these two statements at the very same time, we can't predict what `counter` will contain. It will either contain 4, 5, or 6 after the statements are executed.

Process Coordination & Semaphores

Why does this race condition occur if each of the statements is an independent instruction?

The answer is that - it is not quite true. On the hardware level, every instruction consists of actually three steps: (1) fetch `counter` from memory into a register, (2) increment or decrement the register, and (3) store the new value of `counter` back to memory.

If the instructions from the two processes get interleaved, the result of the execution of the instructions is unpredictable (as the figure on the following slide shows,) and depends on when the instructions are executed by the operating system. Race conditions are notoriously difficult to identify and debug, because by their very nature they only occur on rare occasions, and only when the timing is just exactly right.

The solution is to only allow one process at a time to manipulate the value inside `counter`, and we will see several ways of achieving this.

Process Coordination & Semaphores

Producer:

$$register_1 = \texttt{counter}$$
$$register_1 = register_1 + 1$$
$$\texttt{counter} = register_1$$

Consumer:

$$register_2 = \texttt{counter}$$
$$register_2 = register_2 - 1$$
$$\texttt{counter} = register_2$$

Interleaving:

T_0:	*producer*	execute	$register_1 = \texttt{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\texttt{counter} = register_2$	$\{counter = 4\}$

Interleaving instructions. Taken from Bell, John T. "Process Synchronization." University of Illinois, Chicago.

Process Coordination & Semaphores

Any questions?

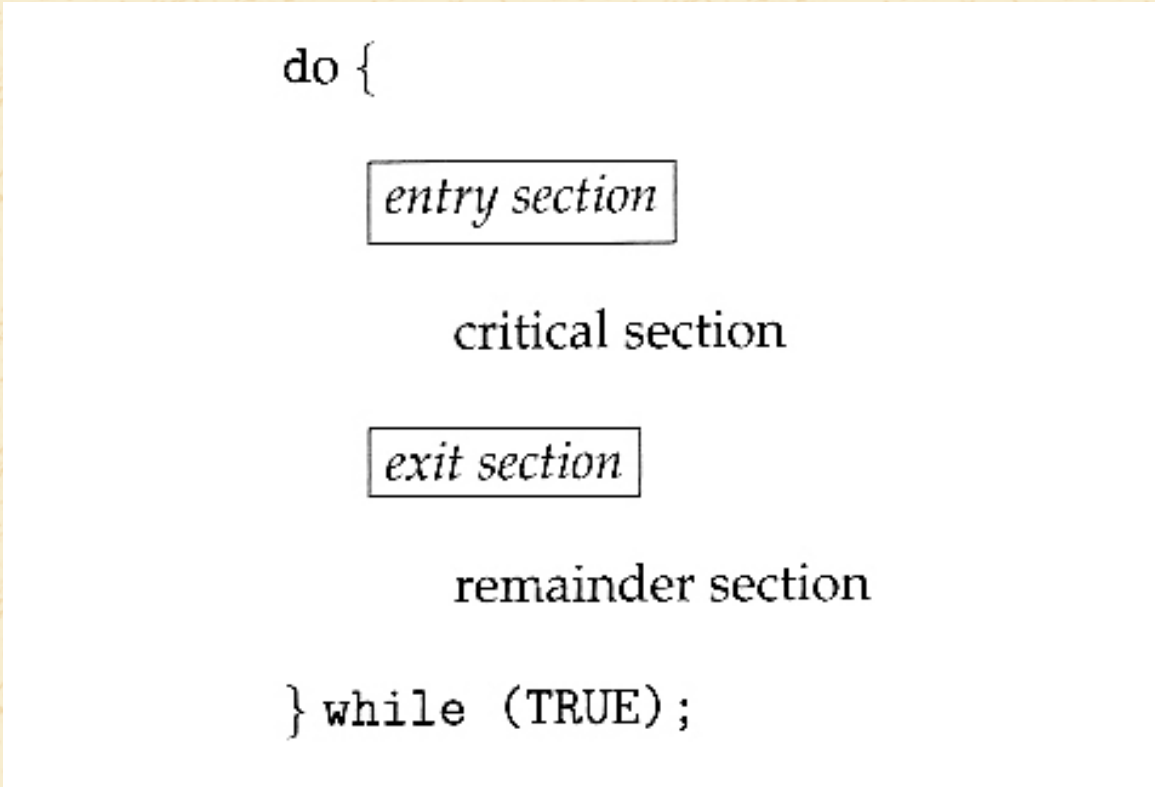
The Critical Section Problem

The Producer-Consumer problem that we discussed is a particular example of a more general problem called the **critical section problem**, which is the need to define a set of rules that the coorporating processes should follow when attempting to change variables that are shared by the processes. The lines of code that modify the value of at least one shared variable are called the **critical section**. The goal is to find a way to alert all the processes that one of them is currently executing code inside a critical section so that the other processes must wait for the current process to finish its work.

The code preceding the critical section, and which controls access to the critical section, is called the **entry section**. It acts like a carefully controlled locking door. The code following the critical section is called the **exit section**. It generally releases the lock on someone else's door, or at least lets the world know that they are no longer in their critical section.

The rest of the code not included in either the critical section or the entry or exit sections is termed the **remainder section**.

The Critical Section Problem



General structure of a typical process. Taken from Bell, John T. "Process Synchronization." University of Illinois, Chicago.

The Critical Section Problem

For a solution to the critical section problem to work, it has to satisfy the following 3 conditions:

1. **Mutual Exclusion**: Only one process at a time can execute code in their critical section.
2. **Progress**: If no process is currently executing their critical section but one or more want to, then only the processes *not* in their remainder sections can participate in the decision of whom will execute the code first, and the decision cannot be postponed indefinitely. (I.e., processes cannot wait forever to get into their critical sections.)
3. **Bounded Waiting**: There exists a limit on the number of times that other processes can get into their critical sections after a process made a request to enter their critical section and before that request is granted. (I.e., a process requesting entry into their critical section will get a turn eventually, and there is a limit as to how many other processes get to go first.)

We assume that all processes proceed at a non-zero speed, but no assumptions can be made regarding the relative speed of one process versus another.

Peterson's Solution

Peterson's Solution is a classical software-based solution to the critical section problem. However, due to the way modern computer architectures perform basic machine-language instructions, such as `load` from and `store` to memory, there are no guarantees that Peterson's solution will work correctly on such architectures, but we still want to show this solution's good algorithmic thinking and consideration of the complexities involved in the solution.

Peterson's solution is based on two processes, P_0 and P_1, which alternate between their critical sections and remainder sections. More generally, P_i is the current process, and P_j (where $j = 1 - i$) is the 'other' process. The solution requires two shared data items:

1. **`int turn`**: Indicates whose turn it is (who is ***allowed***) to enter into the critical section. If `turn == i`, then process i is allowed into their critical section.
2. **`boolean flag[2]`**: Indicates when a process ***wants to*** enter into their critical section. When process i wants to enter their critical section, it sets `flag[i]` to `TRUE`.

Peterson's Solution

```
do {

    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

Peterson's Solution. Taken from Bell, John T. "Process Synchronization." University of Illinois, Chicago.

Peterson's Solution

In the above pseudocode, process `i` wants to enter their critical section (`flag[i] = TRUE`), but first lets process `j` enter their critical section if its wants to (`turn = j`). After process `j` executes the code if wanted (`while (flag[j] && turn = j);`), process `i` is finally allowed to enter their critical section. Once process `i` completes the execution inside the critical section, it lowers the flag (`flag[i] = FALSE`).

To prove that the solution is correct, we must examine the three conditions listed previously:

1. **Mutual Exclusion**: If one process is executing their critical section when the other wishes to do so, the second process will become blocked by the `flag` of the first process. If both processes attempt to enter at the same time, the last process to execute `turn = j` will be blocked. So, this condition holds.
2. **Progress**: The shared variable `turn` assures that only one process at a time can be blocked, and the `flag` variable allows one process to release the other when exiting their critical section. Since one of the processes will continue executing, there is progress and this condition holds.
3. **Bounded Waiting**: As each process enters their entry section, they set the `turn` variable to be the other processes turn. Since no process ever sets it back to their own turn, this ensures that each process will have to let the other process go first *at most one time* before it becomes their turn again, so this condition holds.

Hardware Solutions

1. One simple solution to the critical section problem is to prevent a process from being interrupted while executing their critical section. Unfortunately, this doesn't work well in multiprocessor environments, due to the difficulties in disabling and the re-enabling interrupts on all CPUs. There is also a question as to how this approach affects timing if the clock interrupt is disabled.
2. Another approach is for hardware to provide certain atomic operations. **Atomic** instructions are those that are guaranteed to operate as a single hardware instruction, without interruption. One such operation is the "Test and Set" (see next slide,) which simultaneously (1) sets a boolean lock variable and (2) returns its previous value.
3. Another variation on the test-and-set is an atomic swap of two booleans (see the slide after the next.)

The above examples don't guarantee bounded waiting. If several processes are trying to get into their critical sections, there is no guarantee of what order they will enter, and some process might turn to wait forever until they got their turn in the critical section. This is because, due to the relative rates of the processes, a very fast process could theoretically release the lock, speed through their remainder section, and re-lock the lock before a slower process got a chance.

Hardware Solutions

```
boolean TestAndSet(boolean *target) {
   boolean rv = *target;
   *target = TRUE;
   return rv;
}
```

Figure 5.3 The definition of the `TestAndSet()` instruction.

```
do {
   while (TestAndSetLock(&lock))
      ; // do nothing

      // critical section

   lock = FALSE;

      // remainder section
}while (TRUE);
```

Figure 5.4 Mutual-exclusion implementation with `TestAndSet()`.

TestAndSet() function. Taken from Bell, John T. "Process Synchronization." University of Illinois, Chicago.

Hardware Solutions

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

Figure 5.5 The definition of the `compare_and_swap()` instruction.

```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
} while (true);
```

Figure 5.6 Mutual-exclusion implementation with the `compare_and_swap()` instruction.

CompareAndSwap() function. Taken from Bell, John T. "Process Synchronization." University of Illinois, Chicago.

Hardware Solutions

4. One can change the "Test and Set" solution such that is meets the bounded waiting condition, using two shared data structures: `boolean lock` and `boolean waiting[N]`, where N is the number of processes wanting to execute their critical sections (see pseudocode on the next slide.)

   The key feature of this method is that when exiting a critical section, the exiting process does not just unlock the critical section and allows any other process execute the section. Rather, it first looks in an orderly progression (starting with the next process on the list) for a process that has been waiting, and if it finds one, then it releases that particular process from its waiting state, without unlocking the critical section, thereby allowing a specific process into the critical section while continuing to block all the others.

   Unfortunately, hardware level locks are especially difficult to implement in multi-processor architectures.

Hardware Solutions

```
do {
   waiting[i] = TRUE;
   key = TRUE;
   while (waiting[i] && key)
      key = TestAndSet(&lock);
   waiting[i] = FALSE;

      // critical section

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = FALSE;
   else
      waiting[j] = FALSE;

      // remainder section
}while (TRUE);
```

Bounded-waiting with TestAndSet(). Taken from Bell, John T. "Process Synchronization." University of Illinois, Chicago.

Mutex Locks

Hardware-based solutions are not accessible to application programmers and are highly machine-dependent (different hardware uses different instructions.) As such, most operating systems offer a set of API functions that internally implement such solutions.

The simplest of these tools is the **mutex lock** or **mutexes**, short for mutual exclusion. This solution consists of two functions, which we generically call **`acquire()`** (to acquire a lock prior to entering a critical section) and **`release()`** (to release the lock when exiting the critical section.)

Just as with hardware locks, the acquire step will block the process if the lock is in use by another process, and both the acquire and release operations are atomic.

Example: Linux provides the functions `pthread_mutex_lock()` and `pthread_mutex_unlock()` to implement a mutex lock when two or more threads want to access a critical section.

Mutex Locks

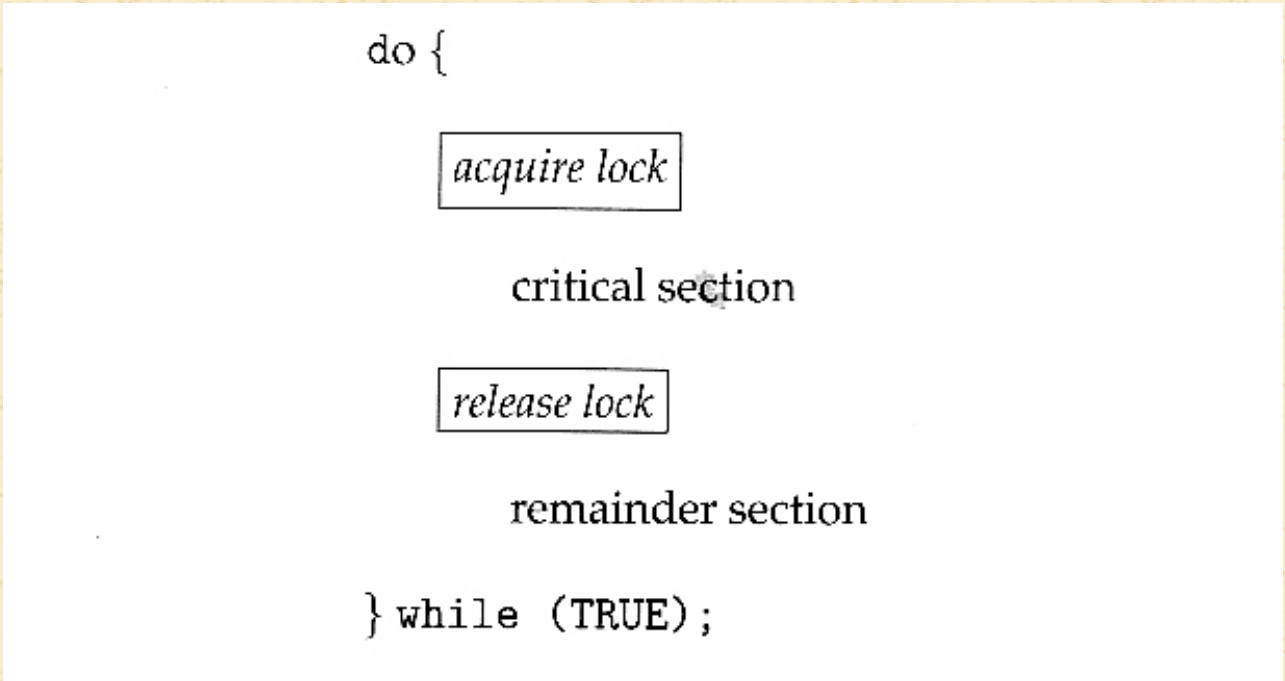
Acquire:

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

Release:

```
release() {
   available = true;
}
```

Possible implementation of acquire() and release(). Taken from Bell, John T. "Process Synchronization." University of Illinois, Chicago.



Using the Mutex method. Taken from Bell, John T. "Process Synchronization." University of Illinois, Chicago.

Semaphores

A more robust alternative to simple mutexes is to use **semaphores**, which are integer variables for which only two atomic operations are defined: `wait()` and `signal()`.

Wait:

```
wait(S) {
    while S <= 0
       ; // no-op
    S--;
}
```

Signal:

```
signal(S) {
    S++;
}
```

Possible implementation of wait() and signal(). Taken from Bell, John T. "Process Synchronization." University of Illinois, Chicago.

```
do {
   waiting(mutex);

      // critical section

   signal(mutex);

      // remainder section
}while (TRUE);
```

Using the Semaphores method. Taken from Bell, John T. "Process Synchronization." University of Illinois, Chicago.

Semaphores

In practice, semaphores can take on one of two forms:

1. **Binary semaphores** can take on one of two values, 0 or 1. They can be used to solve the critical section problem and can be used as mutexes on systems that do not provide a separate mutex mechanism. The right-hand image on the previous slide is an example of such a binary semaphore.
2. **Counting semaphores** can take on some non-zero integer value N (`0`, `1`, `2`, `3`, ..., N), and are usually used to count the number of remaining limited resources. The counter is initialized to the number of such resources available in the system, and whenever the counting semaphore is greater than zero, a process can enter a critical section and use one of the resources. Otherwise, the process blocks until another process frees up a resource and increments the counting semaphore with a `signal()` call.
   A good real-life analogy is when people are trying to enter a room, elevator, etc., but there is space only for a limited number of individuals at a time. When the room's capacity is full, the other people must wait outside until someone exits the room.
   A binary semaphore is just a special case of a counting semaphore when $N = 1$.

Monitors

Semaphores can be very useful for solving concurrency problems, but only if programmers use them properly.

If even one process fails to abide by the proper use of semaphores, either accidentally or deliberately, then the whole system breaks down. Since concurrency problems are by definition rare events, the problematic code may easily go unnoticed and be hard to debug.

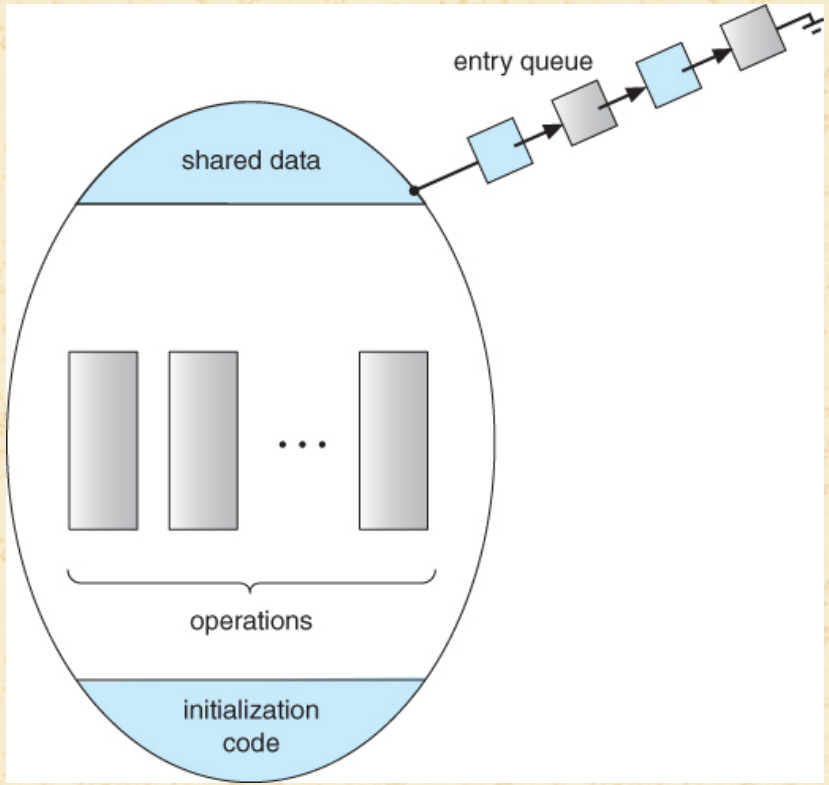
For this reason, a higher-level language construct has been developed, called **monitors**, which can replace the use of semaphores. In the `Monitor` class, all the data is private and only one function within any given `Monitor` object may be active at the same time. An additional restriction is that `Monitor` methods may only access the shared data within the monitor and any data passed to them as parameters. That is, they cannot access any data external to the monitor.

Java and C# ("C sharp"), two object-oriented langauges, offer `Monitor` APIs.

Monitors

```
monitor monitor name
{
   // shared variable declarations

   procedure P1 ( . . . ) {
      . . .
   }

   procedure P2 ( . . . ) {
      . . .
   }

         .
         .
         .
   procedure Pn ( . . . ) {
      . . .
   }

   initialization code ( . . . ) {
      . . .
   }
}
```

Possible implementation of a Monitor class. Taken from Bell, John T. "Process Synchronization." University of Illinois, Chicago.



Schematic view of a monitor. Taken from Bell, John T. "Process Synchronization." University of Illinois, Chicago.

Monitors

Any questions?

Process Coordination & Semaphores: Sources

Bell, John T. "Process Synchronization." *University of Illinois, Chicago*, Accessed 11 July 2022. URL: https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/5_Synchronization.html

Chapter 4: Synchronization and Deadlocks (pp. 93-124).
Hailperin, Max (2019). Operating Systems and Middleware: Supporting Controlled Interaction. (Revised edition 1.3.1) S.l.: Gustavus Adolphus College. This work is licensed under the Creative Commons Attribution-ShareAlike3.0 Unported License. URL: https://gustavus.edu/mcs/max/os-book/osm-rev1.3.1.pdf#chapter.4