

# UNIX Input/Output Buffering

When a C/C++ program begins execution, the operating system environment is responsible for opening three files and providing file pointers to them:

- `stdout`—standard output
- `stderr`—standard error
- `stdin`—standard input

For C programs these are declared in `<stdio.h>` with the the library routine `fprintf()` is built on the system call `write()` for formatting output and error messages. It buffers messages. The library routine `scanf()` is used for reading formatted input. It is built on the system call `read()`.

For C++ programs, these file pointers are accessed through the objects `cout`, `cerr` of the class `ostream` and `cin` of class `istream`.

Examples:

```
fprintf(stdout, "The output is %d\n", sum);
printf("The output is %d\n", sum); /* equivalent to above */
cout << "The output is " << sum << '\n';
```

What is the output of the following program?

```
#include <iostream.h>
#include <unistd.h>

main(int argc, char *argv[])
{
    cout << "hello there, ";
    if (fork())
        cout << "this is parent process " << getpid() << '\n';
    else
        cout << "this is child process " << getpid() << '\n';
}
```

Because the output is buffered and then duplicated on the `fork()`, the output is

```
hello there, this is parent process 8073
hello there, this is child process 8074
```

Buffer will flush when the process exits or when the routine *fflush()* is used in C. In C++ use *cout.flush()* or *endl*. Be sure to flush when debugging as print statements may be executed, but the output cached.

# Process Coordination

*Process coordination* or *concurrency control* deals with *mutual exclusion* and *synchronization*.

*Mutual exclusion*—ensure that two concurrent activities do not access shared data (resource) at the same time, *critical region*—set of instructions that only one process can execute.

*Synchronization*—using a condition to coordinate the actions of concurrent activities. A generalization of mutual exclusion. One process waits for another.

When considering process coordination, we must keep in mind the following situations:

1. *Deadlock* occurs when two activities are waiting each other and neither can proceed. For example:

Suppose processes A and B each need two resources to continue, but only one resource has been assigned to each of them. If the system has only 2 such resources, neither process can ever proceed.

2. *Starvation* occurs when a blocked activity is consistently passed over and not allowed to run. For example:

Consider two cpu bound jobs, one running at a higher priority than the other. The lower priority process will never be allowed to execute. As we shall see, some synchronization primitives lead to starvation.

## Mutual Exclusion

A solution to the mutual exclusion problem should satisfy the following requirements:

**mutual exclusion** — never allow more than one process to execute in a critical section simultaneously

**environment independent** — no assumptions on relative process speeds or number of processors

**resources shared only in critical region** — no process stopped outside of the critical region should block other processes

**bounded waiting** — once a process has made a request to enter a critical region, there must be a bound on the number of times that other processes are allowed to enter the critical sections before the request is granted.

## Critical Sections

A *critical section (region)* is a group of instructions that must be executed as a unit while other activity is excluded. Consider two processes A and B:

```
int balance = 0; /* global shared variable (not Unix!) */
ProcessA()
{
    Deposit(10);
    cout << "Balance is " << balance << '\n';
}

ProcessB()
{
    Deposit(10);
    cout << "Balance is " << balance << '\n';
}

Deposit(int deposit)
{
    int newbalance; /* local variable */
    newbalance = balance + deposit;
    balance = newbalance;
}
```

If *balance* starts with an initial value of 0, what will its ending value be?

- 20, (If we are lucky.)
- 10, otherwise

In our example, the statements in *Deposit()* are a critical section; once execution in the critical section begins, we must insure that it completes before any other activity executes that critical section.

Because *newbalance* is a variable local to *Deposit()*, the variable will be different for the two processes.

## Race Condition

Where the result depends on the relative timing of the processes.

How to ensure that *Deposit()* is executed as a critical region? Need primitives to enforce execution as a critical region: *BeginRegion()* and *EndRegion()*. With mutual exclusion the program is

```
int balance = 0; /* global shared variable (not Unix!) */
ProcessA()
{
    Deposit(10);
    cout << "Balance is " << balance << '\n';
}

ProcessB()
{
    Deposit(10);
    cout << "Balance is " << balance << '\n';
}

Deposit(int deposit)
{
    int newbalance; /* local variable */
    BeginRegion(); /* enter critical region */
    newbalance = balance + deposit;
    balance = newbalance;
    EndRegion(); /* exit critical region */
}
```

## Disabling Interrupts (and Context Switching)

One of the simplest ways to enforce mutual exclusion is to:

1. Disable interrupts at the start of the critical section.
2. Ensure that the activity doesn't give up the CPU before completing the critical region (e.g., don't context switch by calling *resched* or any routine that does).
3. Re-enable interrupts at the end of the critical section.

```
BeginRegion()  
{  
    DisableInterrupts();  
}
```

```
EndRegion()  
{  
    EnableInterrupts();  
}
```

Disabling interrupts has the following disadvantages:

1. One must be careful not to disable interrupts for too long; devices that raise interrupts need to be serviced!
2. Disabling interrupts prevents all other activities, even though many may never execute the same critical region. Disabling interrupts is like using a sledge hammer; a powerful tool, but bigger than needed for most jobs.
3. Programmer must remember to restore interrupts when leaving the critical section (may not have user level access)
4. The programmer must be careful about nesting. Activities that disable interrupts must restore them to their previous settings. In particular, if interrupts are already disabled before entering a critical region, they must remain disabled after leaving the critical region. Code in one critical region may call a routine that executes a different critical region.
5. Technique is ineffective on multiprocessor systems, where multiple processes may be executing in *parallel*. Parallel processing execute multiple processes in parallel. This differs from multiprogramming where only one process can actually execute at one time; the “parallel” execution is simulated.

## Busy Waiting

Another approach is to define a *boolean (lock)* variable that is set to “true” if some activity is currently executing the critical region, “false” otherwise. One (shortsighted!) solution might be:

```
#define TRUE 1
#define FALSE 0
int mutex = 0;          /* also called lock variable */

BeginRegion()          /* Loop until safe to enter */
{
    while (mutex)
        ; /* do nothing until FALSE */
    mutex = TRUE;
}

EndRegion()            /* Exit critical section */
{
    mutex = FALSE;
}

BeginRegion();
    /* code for critical section */
EndRegion();
```

Code for *BeginRegion()* compiles to

```
        lw      $14, mutex      ; load mutex into register 14
        beq     $14, 0, $33     ; branch to $33 if mutex is 0 (FALSE)
$32:    lw      $15, mutex      ; load mutex into register 15
        bne     $15, 0, $32     ; branch to $32 if mutex is 1 (TRUE)
$33:    li      $24, 1          ; load a 1 (TRUE) into register 24
        sw     $24, mutex      ; store value into mutex
```

Do our routines work correctly? No! A process that finds *mutex* set to FALSE may get past the *bne* statement but be rescheduled *before* it actually changes the value of *mutex*. While it sits on the ready list, another process that test the value of *mutex* will find its value set to FALSE.

Solution: we need a mechanism for *atomically* fetching and setting the value of *mutex*. That is, we want to fetch the value, and if it is FALSE, set it to TRUE, all in one instruction. If such a step takes more than one instruction, a process could be interrupted or rescheduled before it has a chance to finish the job.

## Software Solutions

Strict alternation.

Peterson's solution (see Tanenbaum).

## Test-and-Set Lock Instruction

Most machine provides provide an atomic “test and set” instruction for this purpose. Most test-and-set instructions have the following semantics (use C++ call-by-reference semantics to express atomic operation):

```
// defn for atomic operation
int test_and_set(int &var, int value)
{
    int temp;

    temp = var;    // remember old value of variable
    var = value;  // store new value
    return(temp); // return old value of variable
}
```

*BeginRegion* and *EndRegion* can now be rewritten as:

```
BeginRegion()    /* Loop until safe to enter */
{
    while (test_and_set(mutex, TRUE)) ;
        /* Loop until return value is false */;
}

EndRegion()
{
    mutex = FALSE;
}
```

Also called a “spin lock”.

Advantages of above approach:

1. It works!
2. It works for any number of processors (used by multiprocessor)

Disadvantage of above approach: CPU busy-waits until it can enter the critical region, wasting resources.

## Synchronization

```
int n = 0; /* shared by all processes */

main()
{
    int producer(), consumer();
    CreateProcess(producer);
    CreateProcess(consumer);
    /* wait until done */
}

producer() // "produce" values of n
{
    int i;
    for (i=0; i<2000; i++)
        n++; // increment n
}

consumer() // "Consume" and print values of n
{
    int i;
    for (i=0; i<2000; i++)
        cout << "n is " << n << '\n'; // print value of n
}
```

Assume goal is for the consumer to print each value produced. What output does the program generate?

- impossible to predict — depends on how processes are scheduled
- the number 0, 2000 times, if *consumer()* executes before *producer()* begins
- the number 2000, 2000 times, if *producer()* executes before *consumer()* begins
- increasing values of n, some values printed many times, others not printed at all

Problem: producer and consumer need to *synchronize* with each other.

## Semaphores

*Semaphores* are an abstract entity provided by an operating system (not the hardware).  
Semaphores (counting):

- are named by a unique semaphore id
- consist of a tuple (id, count, queue), where count is an integer and queue is a list of processes.
  - a non-negative count always means that the queue is empty
  - a count of negative  $n$  indicates that the queue contains  $n$  waiting processes.
  - a count of positive  $n$  indicates that  $n$  resources are available and  $n$  requests can be granted without delay.
- $sem = semcreate(val)$  — creates a semaphore with the given initial value
- $semdelete(sem)$  — delete a semaphore
- $wait(sem)$  — decrement the semaphore count. if negative, suspend the process and place in queue. (Also referred to as  $P()$ , *down* in literature.)
- $signal(sem)$  — increment the semaphore count, allow the first process in the queue to continue. (Also referred to as  $V()$ , *up* in literature.)

First introduced by Dijkstra (1965) as binary semaphores and the operations were P (wait) and V (signal).

Can be used to implement mutual exclusion:

```
int sem;

sem = semcreate(1);

BeginRegion()
{
    wait(sem);
}

EndRegion()
{
    signal(sem);
}
```

## Example with Fix

```
int psem, csem; /* semaphores */
int n = 0;
main()
{
    int producer(), consumer();
    csem = semcreate(0);
    psem = semcreate(1);
    CreateProcess(producer);
    CreateProcess(consumer);
    // wait until done
}
producer()
{
    int i;
    for (i=0; i<2000; i++) {
        wait(psem);
        n++; // increment n by 1
        signal(csem);
    }
}
consumer()
{
    int i;
    for (i=0; i<2000; i++) {
        wait(csem);
        cout << "n is " << n << '\n'; // print value of n
        signal(psem);
    }
}
```

Now consumer prints all values of n (1-2000).

Points to note:

- Why not a single semaphore like mutual exclusion? Because we need to *synchronize* the two processes!!
- If *consumer()* starts executing first, it will block, because the count of semaphore *csem* will be -1.
- If *producer()* starts executing first, it will increment the value, but then block when it issues *wait* on *psem* a second time.
- When *consumer()* signals *psem*, its count will go to zero, and the suspended process *producer()* will be moved to the ready list.

Advantages of semaphores:

- Processes do not busy wait while waiting for resources. While waiting, they are in a “suspended” state, allowing the CPU to perform other chores.
- Works on (shared memory) multiprocessor systems.
- User controls synchronization.

Disadvantages of semaphores:

1. can only be invoked by processes—not interrupt service routines because interrupt routines cannot block
2. user controls synchronization—could mess up.

## UNIX version (with our routines)

```
// prodcons.C
#include <iostream.h>
#include <unistd.h>
#include <stdlib.h>
#include "sem.h"

int CreateProcess(void (*)());      /* func. prototype */

int psem, csem; /* semaphores */
int *pn;
main()
{
    void producer(), consumer();

    pn = (int *)shmcreate(sizeof(int));
    *pn = 0;
    csem = semcreate(0);
    psem = semcreate(1);
    CreateProcess(producer);
    consumer();          // let parent be the consumer
    semdelete(csem);
    semdelete(psem);
    shmdelete((char *)pn);
}
void producer()
{
    int i;
    for (i=0; i<5; i++) {
        semwait(psem);
        (*pn)++; // increment n by 1
        semsignal(csem);
    }
}
void consumer()
{
    int i;
    for (i=0; i<5; i++) {
        semwait(csem);
        cout << "n is " << *pn << '\n'; // print value of n
        semsignal(psem);
    }
}
```

```
int CreateProcess(void (*pFunc)())
{
    int pid;

    if ((pid = fork()) == 0) {
        (*pFunc)();
        exit(0);
    }
    return(pid);
}
```

## More on Semaphores

How does the Operating System implement semaphores? Like the process table, it disables interrupts (or uses spin lock on multiprocessor) when manipulating the semaphore table.

Look at Fig. 2-24 in text. Why the mutex semaphore? Because there could be multiple producers and consumers and the buffer size is greater than one.

## Summarizing Semaphores

What are the number of semaphores and the initial counts for each of the following situations:

1. mutex for two processes: one semaphore with init count of 1
2. mutex for three processes: same
3. one producer/one consumer, single shared value: two sems, count of 0 and 1
4. one producer/one consumer, n shared values: two sems, count of 0 and n
5. two producers/two consumers, single shared value: two sems, count of 0 and 1
6. two producers/two consumers, n shared values: three sems, count of 0, n and 1 (for mutex).