

Exercises

6.1 In Section 6.4, we mentioned that disabling interrupts frequently can affect the system's clock. Explain why this can occur and how such effects can be minimized.

6.2 What is the meaning of the term *busy waiting*? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer.

6.3 Explain why spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor systems.

6.4 Show that, if the `wait()` and `signal()` semaphore operations are not executed atomically, then mutual exclusion may be violated.

6.5 Illustrate how a binary semaphore can be used to implement mutual exclusion among n processes.

6.6 Race conditions are possible in many computer systems. Consider a banking system that maintains an account balance with two functions: `deposit(amount)` and `withdraw(amount)`. These two functions are passed the `amount` that is to be deposited or withdrawn from the bank account balance. Assume that a husband and wife share a bank account. Concurrently, the husband calls the `withdraw()` function, and the wife calls `deposit()`. Describe how a race condition is possible and what might be done to prevent the race condition from occurring.

6.7 The pseudocode illustrates the basic `push()` and `pop()` operations of an array-based stack. Assuming that this algorithm could be used in a concurrent environment, answer the following questions:

a. What data have a race condition?

b. How could the race condition be fixed?

```
push(item) {
    if (top < SIZE) {
        stack[top] = item;
        top++;
    }
    else
        ERROR
}

pop() {
    if (!is_empty()) {
        top--;
        return stack[top];
    }
    else
        ERROR
}

is_empty() {
    if (top == 0)
        return true;
    else
        return false;
}
```

6.8 Race conditions are possible in many computer systems. Consider an online auction system where the current highest bid for each item must be maintained. A person who wishes to bid on an item calls the `bid(amount)` function, which compares the amount being bid to the current highest bid. If the amount exceeds the current highest bid, the highest bid is set to the new amount. This is illustrated below:

```
void bid(double amount) {
    if (amount gt; highestBid)
        highestBid = amount;
}
```

Describe how a race condition is possible in this situation and what might be done to prevent the race condition from occurring.

6.9 The following program example can be used to sum the array `values` of size N elements in parallel on a system containing N computing cores (there is a separate processor for each array element):

```
for j = 1 to log2(N) {
    for k = 1 to N {
        if ((k + 1) % pow(2, j) == 0) {
            values[k] += values[k - pow(2, (j-1))]
        }
    }
}
```

This has the effect of summing the elements in the array as a series of partial sums, as shown in Figure [Figure E6.17](#). After the code has executed, the sum of all elements in the array is stored in the last array location. Are there any race conditions in the above code example? If so, identify where they occur and illustrate with an example. If not, demonstrate why this algorithm is free from race conditions.

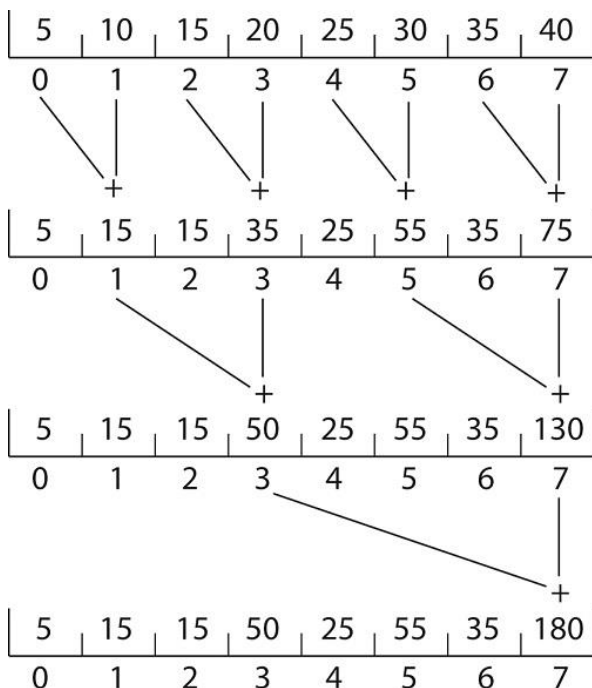


Figure E6.17 Summing an array as a series of partial sums for [Exercise 6.14](#).

6.10 The `compare_and_swap()` instruction can be used to design lock-free data structures such as stacks, queues, and lists. The program example shown in [Figure E6.18](#) presents a possible solution to a lock-free stack using CAS instructions, where the stack is represented as a linked list of `Node` elements with `top` representing the top of the stack. Is this implementation free from race conditions?

```

typedef struct node {
    value_t data;
    struct node *next;
} Node;

Node *top; // top of stack

void push(value_t item) {
    Node *old_node;
    Node *new_node;

    new_node = malloc(sizeof(Node));
    new_node->data = item;
    do {
        old_node = top;
        new_node->next = old_node;
    }
    while (compare_and_swap(top, old_node, new_node) != old_node);
}

value_t pop() {
    Node *old_node;
    Node *new_node;

    do {
        old_node = top;
        if (old_node == NULL)
            return NULL;
        new_node = old_node->next;
    }
    while (compare_and_swap(top, old_node, new_node) != old_node);

    return old_node->data;
}

```

Figure E6.18 Lock-free stack for [Exercise 6.15](#).

6.11 One approach for using `compare_and_swap()` for implementing a spinlock is as follows:

```

void lock_spinlock(int *lock) {
    while (compare_and_swap(lock, 0, 1) != 0)
        ; /* spin */
}

```

A suggested alternative approach is to use the “compare and compare-and-swap” idiom, which checks the status of the lock before invoking the `compare_and_swap()` operation. (The rationale behind this approach is to invoke `compare_and_swap()` only if the lock is currently available.)

This strategy is shown below:

```

void lock_spinlock(int *lock) {
{
    while (true) {
        if (*lock == 0) {
            /* lock appears to be available */
            if (!compare_and_swap(lock, 0, 1))
                break;
        }
    }
}
}

```

Does this “compare and compare-and-swap” idiom work appropriately for implementing spinlocks? If so, explain. If not, illustrate how the integrity of the lock is compromised.

6.12 Some semaphore implementations provide a function `getValue()` that returns the current value of a semaphore. This function may, for instance, be invoked prior to calling `wait()` so that a process will only call `wait()` if the value of the semaphore is > 0 , thereby preventing blocking while waiting for the semaphore. For example:

```
if (getValue(&sem) > 0)
    wait(&sem);
```

Many developers argue against such a function and discourage its use. Describe a potential problem that could occur when using the function `getValue()` in this scenario.

6.13 The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P_0 and P_1 , share the following variables:

```
boolean flag[2]; /* initially false */
int turn;
```

The structure of process P_i ($i == 0$ or 1) is shown in Figure [Figure E6.19](#). The other process is P_j ($j == 1$ or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem.

```
while (true) {
    flag[i] = true;

    while (flag[j]) {
        if (turn == j) {
            flag[i] = false;
            while (turn == j)
                ; /* do nothing */
            flag[i] = true;
        }
    }

    /* critical section */

    turn = j;
    flag[i] = false;

    /* remainder section */

}
```

Figure E6.19 The structure of process P_i in Dekker's algorithm.

6.14 The first known correct software solution to the critical-section problem for n processes with a lower bound on waiting of $n - 1$ turns was presented by Eisenberg and McGuire. The processes share the following variables:

```
enum pstate {idle, want_in, in_cs};
pstate flag[n];
int turn;
```

All the elements of `flag` are initially `idle`. The initial value of `turn` is immaterial (between 0 and $n-1$). The structure of process P_i is shown in Figure [Figure E6.20](#). Prove that the algorithm satisfies all three requirements for the critical-section problem.

```

while (true) {
    while (true) {
        flag[i] = want_in;
        j = turn;

        while (j != i) {
            if (flag[j] != idle) {
                j = turn;
            } else {
                j = (j + 1) % n;
            }
        }

        flag[i] = in_cs;
        j = 0;

        while ( (j < n) && (j == i || flag[j] != in_cs))
            j++;

        if ( (j >= n) && (turn == i || flag[turn] == idle))
            break;
    }

    /* critical section */

    j = (turn + 1) % n;

    while (flag[j] == idle)
        j = (j + 1) % n;

    turn = j;
    flag[i] = idle;

    /* remainder section */
}

```

Figure E6.20 The structure of process P_i in Eisenberg and McGuire's algorithm.

6.15 Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user-level programs.

6.16 Consider how to implement a mutex lock using the `compare_and_swap()` instruction. Assume that the following structure defining the mutex lock is available:

```

typedef struct {
    int available;
} lock;

```

The value `(available == 0)` indicates that the lock is available, and a value of 1 indicates that the lock is unavailable. Using this struct, illustrate how the following functions can be implemented using the `compare_and_swap()` instruction:

- `void acquire(lock *mutex)`
- `void release(lock *mutex)`

Be sure to include any initialization that may be necessary.

6.17 Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.

6.18 The implementation of mutex locks provided in Section 6.5 suffers from busy waiting. Describe what changes would be necessary so that a process waiting to acquire a mutex lock would be blocked and placed into a waiting queue until the lock became available.

6.19 Assume that a system has multiple processing cores. For each of the following scenarios, describe which is a better locking mechanism—a spinlock or a mutex lock where waiting processes sleep while waiting for the lock to become available:

- The lock is to be held for a short duration.
- The lock is to be held for a long duration.
- A thread may be put to sleep while holding the lock.

6.20 Assume that a context switch takes T time. Suggest an upper bound (in terms of T) for holding a spinlock. If the spinlock is held for any longer, a mutex lock (where waiting threads are put to sleep) is a better alternative.

6.21 A multithreaded web server wishes to keep track of the number of requests it services (known as *hits*). Consider the two following strategies to prevent a race condition on the variable `hits`. The first strategy is to use a basic mutex lock when updating `hits`:

```
int hits;
mutex_lock hit_lock;

hit_lock.acquire();
hits++;
hit_lock.release();
```

A second strategy is to use an atomic integer:

```
atomic_t hits;
atomic_inc(&hits);
```

Explain which of these two strategies is more efficient.

6.22 Consider the code example for allocating and releasing processes shown in [Figure E6.21](#).

```
#define MAX_PROCESSES 255
int number_of_processes = 0;

/* the implementation of fork() calls this function */
int allocate_process() {
    int new_pid;

    if (number_of_processes == MAX_PROCESSES)
        return -1;
    else {
        /* allocate necessary process resources */
        ++number_of_processes;

        return new_pid;
    }
}

/* the implementation of exit() calls this function */
void release_process() {
    /* release process resources */
    --number_of_processes;
}
```

Figure E6.21 Allocating and releasing processes

a. Identify the race condition(s).

b. Assume you have a mutex lock named `mutex` with the operations `acquire()` and `release()`. Indicate where the locking needs to be placed to prevent the race condition(s).

c. Could we replace the integer variable

```
int number_of_processes = 0
```

with the atomic integer

```
atomic_t number_of_processes = 0}
```

to prevent the race condition(s)?