Synchronization Without Busy Waiting

The solutions we've considered up to now, Dekker's and Peterson's algorithms and hardware test-and-set all share an unpleasent attribute. processes waiting to enter the critical section use the CPU to keep checking if they can enter their critical section. The act of using the CPU to repeatedly check for entry to the critical section is called *busy waiting.* Processes that are busy waiting use their whole quantum doing nothing (if they're waiting, the process actually in its critical section can't be advancing...). In general this isn't good.[1]

Not only can busy waiting be inconvenient, on systems with a priority scheduler priority inversions can occur. (Waiting for entry to a critical section is a prime cause of these).

To avoid busy waiting, we'll create resources that enforce mutual exclusion and make them accessible by through the operating system. Processes waiting to enter a critical section will be put in the blocked state, just like they were waiting for I/O, so that other processes can continue doing useful work.

Semaphores

Semaphores are shared counters with blocking semantics. Specifically, A semaphore is a non-negative integer on which 2 operations are defined:

- *P()* or *down()* decrements the semaphore by one.[2] If the semaphore is zero, the process calling P() is blocked until the semaphore is positive again.
- *V()* or *up()* increments the semaphore by one.

P() and V() are *atomic*. A process sees all or none of a P() or V(), and no P() or V() is ever lost. These operations are generally invoked by processes using system calls, and within the kernel, the simple operations of incrementing and decrementing are done with interrupts off to make them atomic. Blocking is accomplished by putting the process on a blocked queue associated with each semaphore. The nachos implementation of semaphores is a good one to look at for the flavor of it.

The Bounded Buffer Problem Using Semaphores

A common situation in operating systems is to have a bounded buffer shared by 2 processes. One (the producer) puts data into the buffer and the other (the consumer) removes it. The buffer has a finite size, so the producer has to wait when it's full, and the consumer has to wait when it's empty. And, of course, all elements added to the queue by the producer must be seen by the consumer exactly once.

This is the common problem that we will use to demonstrate our mutual exclusion systems.

---

[1] One place where busy waiting is less costly is a multiprocessor. If most busy waits are short, it can be cheaper to allow a process on one processor to busy wait than to incur the overhead of a context switch. With more than one processor, two processes can be making progress simultaneously.

[2] The strange letters are because Djkstra, who invented semaphores, named them using the first letters of Dutch words meaning up and down. Operating systems types, fond of strange arcana, keep the names in use.

```
#define N 100   /* Number of free slots */

semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer() {
    item i;

    while (1) {
     i = produce_item();
     P(empty);
     P(mutex);
     insert_new_item(i);
     V(mutex);
     V(full);
    }
}

void consumer() {
    item i;

    while(1) {
     P(full);
     P(mutex);
     i = get_next_item();
     V(mutex);
     V(empty);
     consume_item(i);
    }
}
```

This problem demonstrates 2 kinds of semaphore. A two-valued or *binary semaphore* directly implements mutual exclusion (`mutex` in the example). Multiple-valued or *counting semaphores* can implement synchronized counts of shared resources (`full` and `empty` in the example).

Semaphores are simple and powerful, and easy to misconfi gure. Reverse the P()'s in either the producer or consumer and the program can stop!

Monitors

Monitors are a language construct, if they exist, they're part of the language. In a language that supports monitors, a collection of procedures is declared to be in the monitor, and mutual exclusion is enforced on those procedures - only one process can be executing (or in the ready queue) with its program counter in monitor code. Underneath the covers, this mutual exclusion is implemented with semaphores (or the equivalent), but the calls to P() and V() are placed by the compiler, which is more likely to consistently get it right than people.

Of course, not all procedures will terminate directly; sometimes a critical section will need to wait for another process to do work, like a consumer encountering an empty queue. Monitors need a blocking mechanism, which is provided by *condition variables.* A condition variable isn't really a variable. It doesn't have a value and processes can't examine it. Condition variables are only synchronization points. A process blocks by calling `wait` on a condition. `wait` blocks the process and releases the monitor lock (i.e., increments the monitor semaphore). A process releases another process by using the `signal` system call. There are three possible behaviors of `signal`:

> **Hoare Semantics** Hoare, who invented monitors, suggested that the sleeping process is awakened and run immediately, suspending the `signal`ing process, and restarting it when the `signal`led process leaves the monitor. This gets messy if the `signal led process does another signal.`
>
> **Hansen Semantics:** Hansen made the problem somewhat easier by requiring that after a process makes a `signal` system call, it must immediately exit the monitor. In this case,

process blocked on the condition is simply run immediately, without releasing the monitor (because there is still a process in the monitor - the `signal`led one).

Mesa Semanitcs: Mesa semantics are the simplest of the three to implement - the process to be awakened is put on the ready queue, and run when the scheduler gets to it. The `signal`led process must reacquire the monitor lock in the `wait` system call before returning to the process. Because other intervening processes may run, the process being awakened should also recheck the condition that caused it to `wait`, because some intervening process may have entered the monitor and changed the internal state.

Here's the bounded buffer program solved with monitors:

```
#define N 100   /* Number of free slots */

monitor {
    int count;
    condition full, empty;

    monitor() { empty = 1 ; full = 0; }

    void insert(item i) {
     if (count == N ) wait(full);
     insert_new_item(i);
     count++;
     if ( count == 1 ) signal(empty);
    }

    item get() {
     item i;

     if ( count == 0 ) wait(empty);
     i = get_next_item();
     count--;
     if ( count ==N-1) signal(full);
     return i;
    }
} queue;

void producer() {
    item i;

    while (1) {
     i = produce_item();
     queue.insert(i);
    }
}

void consumer() {
    item i;

    while(1) {
     i = queue.get();
     consume_item(i);
    }
}
```

Monitors are a nice way to write mutlitasking programs, if your language supports them. You can simulate monitors, by putting locks in place manually, but then you're back to placement problems. Still, some systems do so - like nachos.

It used to be that monitors existed in only a few out-of-the-way languages, like Concurrent Euclid and Mesa, but they've received a big shot in the arm lately. Java synchronized classes are essentially monitors. They're not as pure as monitors pushers would like, but there is a full, language-supported implementation there for programmers to use, and it is the supported synchronization mechanism for Java threads.

Message Passing

Monitors and semaphores are basically designed for machines that share one memory - they're data-structure-oriented. As separate computers become more prevalent, another paradigm, *message passing*, has shown itself to be particularly valuable.

Processes in a message-passing system send messages (arbitrary sets of bits) to each other through message queues. Processes can send messages to other processes and receive message (blocking if none are available). There are variations on exactly how precise a message query can be, and what types of messages are allowed, but that's the basic gist of the systems.

The bounded buffer problem looks like this in a message passing system:

```
#define N 100   /* Number of free slots */

void producer() {
    item i;
    message m;
    while (1) {
     i = produce_item();
     receive("consumer",&m);
     m = build_message(i);
     send("consumer",&m);
    }
}

void consumer() {
    item i;
    message m;

    for ( i = 0; i < N ; i++ ) {
     m = empty_message();
     send("producer",&m);
    }

    while(1) {
     m = receive("producer",&m);
     i = parse_message(m);
     consume_item(i);
     m = empty_message();
     send("producer",&m);
    }
}
```

Some problems faced by message passing:

- **Reliability:** if the processes are on different machines, a protocol must insure that messages are delivered in order and none is lost.
- **Addressing:** the processes must have a way of naming each other and arranging for messages to be delivered at all.
- **Authentication:** if processes are on different machines, it is more diffi cult to ensure that the message came from who it purports to have originated it.

Message passing is still a nice paradigm for inter-machine messaging, and has been used extensively by many systems. Mach, which is to be the basis of Apple's new MacIntosh OS, springs to mind.

Equivalence of Methods

There are many other ideas for synchronizing processes: Path Expressions, Event Counters, Serializers, Sequencers, and a bunch I haven't heard of. Fortunately, basically any of the synchronizatiuon primitives can be built from any of the others. This is a powerful truth philosohically (all synchronization systems are different ways of viewing the same essential idea) and practically (user can build a system intuitive to them if the system provides another).

As an example, here's the monitors with semaphores example from Tannenbaum:

```
typedef struct {
    semaphore mutex = 1;
    semaphore wait[N] = 0;
} monitor;

void enter_monitor(monitor M) {
    P(M.mutex);
}

void leave_monitor(monitor M) {
    V(M.mutex);
}

void leave_with_signal(monitor M, int var) {
    V(M.wait[var]);
}

void wait(int var) {
    V(M.mutex);
    P(M.wait[var]);
}
```

(Note that these are Hansen semantics - not the ones you need for Nachos).

A Classic Problem - Dining Philosophers

The Dining Philosophers problem is a classic OS problem that's usuallu stated in very non-OS terms:

> There are *N* philosphers sitting around a circular table eating spaghetti and discussing philosphy. The problem is that each philosopher needs 2 forks to eat, and there are only *N* forks, one between each 2 philosophers. Design an algorithm that the philosophers can follow that insures that none starves as long as each philosopher eventually stops eating, and such that the maximum number of philosophers can eat at once.

Why describe problems this way? Well, the analogous situations in computers are sometimes so technical that they obscure creative thought. Thinking about philosophers makes it easier to think abstractly. And many of the early students of this field were theoreticians who like abstract problems. There are a bunch of named problems - Dining Philosophers, Drinking Philiosophers, Byzantine Generals, etc.

Here's an approach to the Dining Phils[3] that's simple and wrong:

```
void philosopher() {
    while(1) {
    sleep();
    get_left_fork();
    get_right_fork();
    eat();
    put_left_fork();
    put_right_fork();
    }
}
```

If every philosopher picks up the left fork at the same time, noone gets to eat - ever.

Some other suboptimal alternatives:

- Pick up the left fork, if the right fork isn't available for a given time, put the left fork down, wait and try again. (Big problem if all philosophers wait the same time - we get the same failure mode as before, but repeated.) Even if each philosopher waits a different random time, an unlucky philosopher may starve (in the literal or technical sense).

[3] Now that you've seen the problem, you can call them Phil, too.

- Require all philosophers to acquire a binary semaphore before picking up any forks. This guarantees that no philosopher starves (assuming that the semaphore is fair) but limits parallelism dramatically. (FreeBSD has a similar problem with multiprocessor performance).

Here's Tannenbaum's example, that gets maximum concurrency:

```
#define N 5  /* Number of philosphers */
#define RIGHT(i) (((i)+1) %N)
#define LEFT(i) (((i)==N) ? 0 : (i)+1)

typedef enum { THINKING, HUNGRY, EATING } phil_state;

phil_state state[N];
semaphore mutex =1;
semaphore s[N];      /* one per philosopher, all 0 */

void test(int i) {
    if ( state[i] == HUNGRY &&
     state[LEFT(i)] != EATING &&
     state[RIGHT(i)] != EATING ) {state[i] = EATING; V(s[i]);}
}

void get_forks(int i) {
    P(mutex);
    state[i] = HUNGRY;
    test(i);
    V(mutex);
    P(s[i]);
}

void put_forks(int i) {
    P(mutex);
    state[i]= THINKING;
    test(LEFT(i));
    test(RIGHT(i));
    V(mutex);
}

void philosopher(int process) {
    while(1) {
    think();
    get_forks(process);
    eat();
    put_forks(process);
    }
}
```

The magic is in the `test` routine. When a philosopher is hungry it uses `test` to try to eat. If `test` fails, it wiats on a semaphore until some other process sets its state to EATING. Whenever a philosopher puts down forks, it invokes `test` in its neighbors. (Note that `test` does nothing if the process is not hungry, and that mutual exclusion prevents races.)

So this code is correct, but somewhat obscure. And more importantly, it doesn't encapsulate the philosopher - philosophers manipulate the state of their neighbors directly. Here's a version that does not require a process to write another process's state, and gets equivalent parallelism.

```
#define N 5  /* Number of philosphers */
#define RIGHT(i) (((i)+1) %N)
#define LEFT(i) (((i)==N) ? 0 : (i)+1)

typedef enum { THINKING, HUNGRY, EATING } phil_state;

phil_state state[N];
semaphore mutex =1;
semaphore s[N];      /* one per philosopher, all 0 */

void get_forks(int i) {
    state[i] = HUNGRY;
    while ( state[i] == HUNGRY ) {
    P(mutex);
    if ( state[i] == HUNGRY &&
     state[LEFT] != EATING &&
     state[RIGHT(i)] != EATING ) {
        state[i] = EATING;
        V(s[i]);
    }
    V(mutex);
    P(s[i]);
    }
}

void put_forks(int i) {
    P(mutex);
    state[i]= THINKING;
    if ( state[LEFT(i)] == HUNGRY ) V(s[LEFT(i)]);
    if ( state[RIGHT(i)] == HUNGRY) V(s[RIGHT(i)]);
    V(mutex);
}

void philosopher(int process) {
    while(1) {
    think();
    get_forks(process);
    eat();
    put_forks();
    }
}
```

If you really don’t want to touch other processes’ state at all, you can always do the V to the left and right when a philosopher puts down the forks. (There’s a case where a condition variable is a nice interface.)

Tannenbaum discusses a couple of others that are worth looking at.