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[1] 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).
- 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:

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

```
#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.

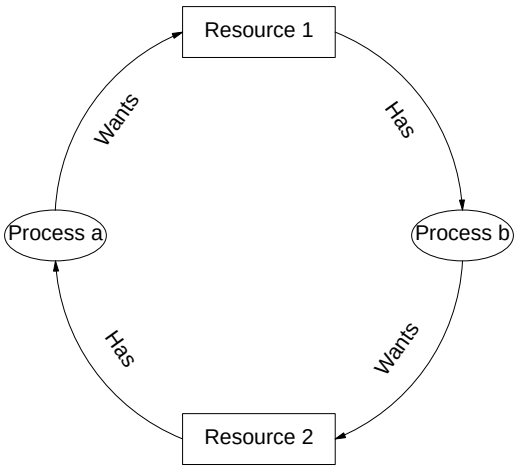
Deadlock

Another danger of concurrent programming is *deadlock*. A race condition is produces incorrect results, where a deadlock results in the deadlocked processes never making any progress. In the simplest form it's a process waiting for a resource held by a second process that's waiting for a resource that the first holds.

Here's an example:

```
#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(mutex);
    P(empty);
    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);
    }
}
```

When the producer arrives at a full queue, it will block on empty while holding mutex, and the consumer will subsequently block on mutex forever. Graphically, this looks like:



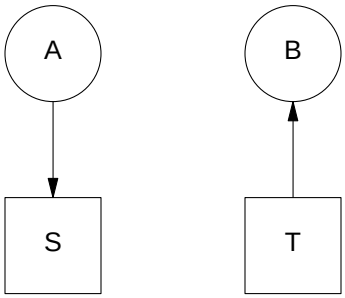
Processes we hopefully understand. Resources are the other parts of the computer system - hardware and software. For the purposes of deadlock, we're concerned with resources that require exclusive control: tape drives, CD-ROM burners, locks & semaphores, process table slots, etc.

4 Conditions

More formally there are 4 conditions for deadlock:

- **Mutual Exclusion:** Resources can only be held by at most one process. A process canniot deadlock on a sharable resource.[2]
- **Hold and Wait:** A process can hold a resource and block waiting for another.
- **No Preemption:** The operating system cannot take a resource back from a process that has it.
- **Circular Wait:** A process is waiting for a resource that another process has which is waiting for a resource that the first has. This generalizes.

If these 4 conditions are necessary and sufficient. (If these four appear, there is a deadlock, if not no deadlock.) The problem is that the chain in a circular wait can be long and complex. We use *resource graphs* to help us visualize deadlocks.



Requesting a resource Holding a resource

All four conditions are can be expressed in the graph:

- **Mutual Exclusion:** Only one one outgoing arrow on a resource.
- **Hold and Wait:** Processes can simultaneously have incoming and outgoing arrows.
- **No Preemption:** Only processes without outgoing lines can delete their incoming lines (only processes not waitig can release resources).
- **Circular Wait:** A directed cycle in the graph.

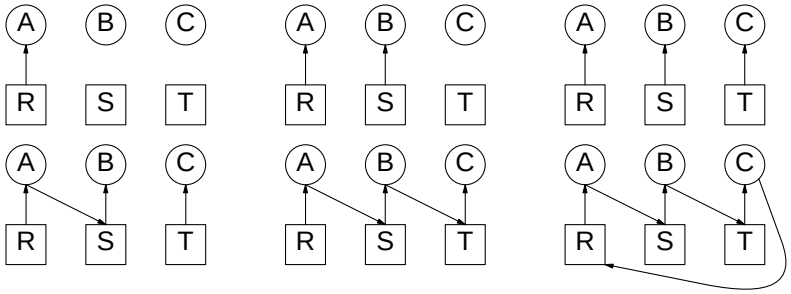
A Deadlock Example

Consider a set of processes making the following requests:

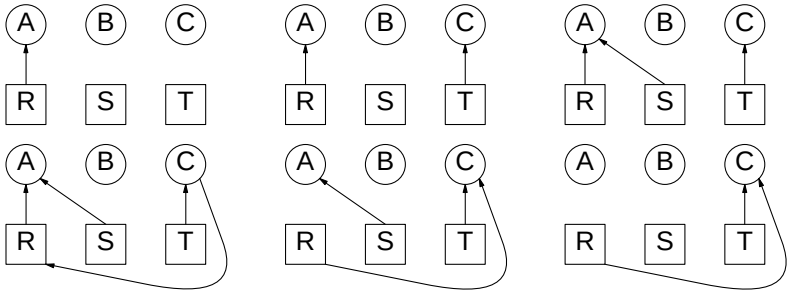
A	B	C
Request R	Request S	Request T
Request S	Request T	Request R
Release R	Release S	Release T
Release S	Release T	Release R

Here's a possible deadlock scenario:

[2] This related to the mutual exclusion used to avoid race conditions. Using locks or semaphores makes the code between the P and the V a mutually exclusive resource.



An OS that can see the future (or that knows the resource needs of the processes) might prevent that deadlock by suspending process B:



Reactions To Deadlock

An OS can react to deadlock one of 4 ways:

- Ignore it
- Detect and Recover from it
- Avoid it (invest effort at run-time to make deadlock impossible)
- Prevent it (short circuit one of the 4 conditions)

Ignoring Deadlock

Running, stable Operating Systems ignore potential deadlocks. In fact, I would venture that there is no production operating system that cannot be deadlocked by pathological behavior.

As an example, UNIX® keeps all possible PCBs in a table. Enough processes that need to great enough subprocesses can deadlock themselves (see Tannenbaum).

This is acceptable because the cost of removing this deadlock from the system is large compared to the chance of it happening. There is a strict overhead/likelyhood/failure cost equation to be balanced here. Note that the equation is different for a life-critical system and for a general purpose operating system.

Detecting Deadlock

Systems detect deadlock by keeping a version of the resource graph avaliable in their memory. This graph is updated on each resource request and release. Asynchronously, graph algorithms that look for cycles are run on the resource graph looking for deadlock.

Alternatively, the system can search for a set of processes that have been idle a long time. This is easier, but can detect deadlocks where none exist. If there are other reasonable means to detect deadlock , this is almost never an appropriate detection mechanism.

Once a deadlock is found, the cycle must be broken. The system can terminate a process or preempt a resource. [...]