Synchronization

Communicating Processes

Despite process abstraction, different computations (processes) sometimes want to communicate. Communication can increase the modularity and robustness of programs (one process for each action, the application can continue if one of the subprocesses fails), and is sometimes an end unto itself.

A good example of both uses of interprocess communication (IPC) is a web server. Modularity is increased by having a separate copy of the server answer each query (the queries are passes from the master browser to a sub browser by IPC). The reason for the server to exist at all is to communicate between a web browser process on one computer with the server on this one.

Lots of seminal work in synchronization was done in a shared memory model, which is the one we'll start with.

- We have several processes timesharing the same single CPU.[1]
- Scheduling is preemptive; between any 2 instructions the running process may be interrupted and context switched for another.
- The processes all have access to a shared are of memory where one process can change a variable that can be seen by all other processes.

Race Conditions

An atomic action is an indivisible action - an action taken by a process than cannot be interrputed by a context switch. When accesses to a shared variable are not atomic, call it a race condition.

Consider the following simple example:

```
shared int x =3;

process_1 () {

    x = x +1;
    print x;
}

process_2 () {

    x = x +1;
    print x;
}
```

Note that although `x = x +1` looks like one statement, it probably compiles into at least 2 and probably three machine instructions. A likely translation is:

```
LOAD r1 x
ADD r1+1 r1
STORE r1 x
```

Because of this, depending on when context switches occur, any of the following sets of output and final values in `x` are possible:

[1] This is called multiprogramming or multitasking. Multiprocessing refers to the contition in which many CPUs, each running a separate program have access to the same memory and peripherals. Multiprocessing synchronization is a related, but slightly harder problem.

Process 1	Process 2	variable
4	5	5
5	4	5
4	4	4

If the program is keeping bank balances, someone's deposit just got lost. If it's controlling chemotherapy, maybe their life.

Critical Sections

Not all code needs to worry about race conditions. Only code that is manipulating shared data needs to worry about them. The most common way to avoid race conditions is to mark the sections of code that are accessing shared data (such parts are called *critical sections* or *critical regions*) and insure that only one process is ever executing code in the critical section or context switched out of code in the critical section. This property is called *mutual exclusion.*

Critical sections are specific to shared data. If `a` and `b` are shared variables that are unrelated to each other, one process can be in a critical section that accesses variable `a` while another process is in a critical section that accesses `b`. Related variables must share mutual exclusion. If a program invariant is that two varibles add to a constant, they must share mutual exclusion. For simplicity, We will talk about ways to enforce mutual exclusion as though there is only one set of related shared variables.

We'll define 2 functions, `enter_region` and `leave_region` that guarantee that at most one process has returned from `enter_region` and not `leave_region`, with the assumption being that a process will call `enter_region` before entering the critical region and `leave_region` immediately after leaving it.

Mutual Exclusion (Simple & Wrong)

```
int flag;

void enter_region(int process) {
    while ( flag == 1 );
    flag = 1;
}

void leave_region(int process) {
    flag = 0;
}
```

That code's wrong, and understanding *why* it's wrong will start you on your way to understanding mutual exclusion. The code fails because `flag` is as much a shared variable as anything in the critical section. The same way that two processes increment `x` in and leave it with incremented once from the point of view of an external observer in our first example, two processes can both see the `flag` as zero before it's set to one, and both enter the critical section.

Alternation

```
int turn;

void enter_region(int process) {
    while ( turn != process ) ;
}

void leave_process(int process) {
    int other = 1 - process;

    turn = other;
}
```

This allows 2 processes alternating access to the critical section. The race condition is avoided because the process of entering the critical section only tests the shared `turn` variable. Missing the other processes set just delays its entry.

This is still of only limited use because strict alternation is required - other access patterns are not supported.

Dekker's Algorithm - a real software-only mutual exclusion algorithm

Dekker's algorithm is the first published software-only, two-process mutual exclusion algotithm.

```
int favored_process;
int interested[2];

void enter_region(int process) {
    int other = 1-process;

    interested[process] = TRUE;
    while ( interested[other] ) {
        if ( favored_process == other ) {
            interested[process] = FALSE;
            while ( favored_process == other );
            interested[process] = TRUE;
        }
    }
}

void leave_region(int process) {
    int other = 1 - process;

    favored_process = other;
    interested[process] = FALSE;
}
```

Dekker adds the idea of a favored process and allows access to either process when the request is uncontested. When there is a conflict, one process is favored, and the priority reverses after successful execution of the critical section. It's worthwhile to trace through the code seeing why each test is in the code and why.

Dekker's algorithm was simplified by Peterson:

```
int turn;
int interested[2];

void enter_region(int process) {
    int other;

    other = 1 - process;
    interested[process] = TRUE;
    turn = process;
    while ( turn == process && interested[other] == TRUE ) ;
}

void leave_region(int process) {
    interested[process] = FALSE;
}
```

This code actually takes advantage of a race condition. When 2 processes try to enter simultaneously, setting `turn` to the entering pid releases the other interested process from the `while` loop. It's a clever trick. Dekker's algorithm was published in 1965, Peterson's in 1981.

Both algorithms can be generalized to multiple processes, although that's beyond the scope of the class.

Perhaps the most amazing thing about software-only mutual exclusion is that it can be done at all in the challenging environment of multitasking.

Hardware Assistence

One way to enforce mutual exclusion is to turn off interrupts while in the critical section. Some OSes, for example, AmigaDOS actually allowed processes to do this. It's a bad idea for several reasons:

- It badly breaks process isolation - a process can prevent other processes from hearing from the disk or timers.
- Turning off interrupts for any period of time is extremely hazardous. The OS needs to field interrupts to keep the machine running, and frequently will be so confused after interrupts are off for a while that it will reboot anyway. (Most hardware will send an NMI (non maskable interrupt) after too long with interrupts disabled).

A more reasonable method of hardware assistence is the *test-and-set instruction*, which makes testing a flag and resetting it's value an *atomic* action. The given memory location is set to the new value and its old value returned in a register without the opportunity for a context switch. This allows a critical section to be coded as:

```
int flag;

void enter_region(int process) {
    int my_flag = test_and_set(flag);

    while ( my_flag == 1 )
    my_flag = test_and_set(flag);
}

void leave_region(int process) {
    flag = 0;
}
```

Because the test and the resetting of the flag is atomic, this system simplifies mutual exclusion by changing the rules - the Wrath of Khan approach.