Interprocess Comunication (IPC)

Processes frequently need to communicate with other processes. For example, in a shell pipeline, the output of the first process must be passed to the second process, and so on down the line. Thus there is a need for communication between processes, preferably in a well-structured way not using interrupts. In the following sections, we will look at some of the issues related to this **InterProcess Communication**, or **IPC**. There are three issues here: 1. How one process can pass information to another 2. Making sure two or more processes do not get in each other's way 3. Proper sequencing when dependendcies are present From these three, the last two also apply to threads (first doens't has threads share the same address space). **The solutions given for processes are also valid for threads.**

1. Race Conditions

Situations, where two or more processes are reading or writing some shared data and the final result depends on who run precisely when, are called **race conditions.**

2. Critical regions

How do we avoid race conditions? The key is to find some way to prohibit more than one process from reading and writing the shared data at the same time. Put in other words, what we need is **mutual exclusion**, a way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing. The part of the program is where shared memory is accessed is called **critical region** or **critical section**. We need four conditions to hold to have a good solution:

1. No two processes may be simultaneously inside their critical regions
2. No assumptions may be made about the speeds or the number of CPUs
3. No process running outside its critical region may block any process
4. No process should have to wait forever to enter its critical region

3. Mutual Exclusion with Busy Waiting

Disabling Interrupts

On a single-processor system, the simplest solution is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it. The CPU is only switched from process to process as a result of clock or other interrupts; therefore, by disabling them, the CPU will not be switched to another process. However, giving user processes the power to turn off interrupts is unwise. What if it never turns them on again? Furthermore, if the

system is a multiprocessor, disabling interrupts only affects the CPU that executed the disable instruction.

Lock Variables

Let us look for a software solution. Consider having a single, shared (lock) variable, initially 0. When a process wants to enter its critical region, it first tests the lock. If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0. Thus, a 0 means that no process is in its critical region and a 1 means that some process is in its critical region. This leads us to the same fatal flaw. Two processes may check the variable at the same time and read it as 0; this will lead to two processes setting the lock variable to 1 and be in their respective critical region at the same time.

Strict alternation

The integer variable turn, initially 0, keeps track of whose turn it is to enter the critical region and examine or update the shared memory. Continuously testing a variable until some value appears is called **busy waiting**. It should usually be avoided, since it wastes CPU time. Only when there is a reasonable expectation that the wait will be short is busy waiting used. A lock that uses busy waiting is called a **spin lock**. This implementation may block a process without any of them being in their critical region: if one process is slower, it may cause one other process to enter busy waiting until its turn. This violates condition 3 aforementioned, therefore it is not a serious candidate for our problem.

Peterson's Solution

Before entering the critical region, each process calls *enter_region* with its own process number as parameter. This will cause it to wait, if need be, until it is safe to enter. After it has finished with the shared variables, the process calls *leave_region* to indicate that it is done and to allow the process to enter, if it so desires. Now consider that both processes call *enter_region* almost simultaneously. Both will store their process number in turn. Whichever store is done last is the one that counts; the first one is overwritten and lost. Suppose that process 1 stores last, so turn is 1. When both processes come to the while statement, process 0 executes it zero timer and enters its critical region. Process 1 loops and does not enter its critical region until process 0 exits its critical region.

TSL Instruction

Test and Set Lock works as follows. It reads the contents of the memory word *lock* into register RX and then stores a nonzero value at the memory address *lock*. The operations of reading the word and storing into it are guaranteed to be indivisible - no other processor can access the memory word until the instruction is finished. The CPU executing the TSL instruction locks the memory bus to

prohibit other CPUs from accessing memory until it is done.

4. Sleep and Wakeup

The problem with TSL or XCHG is that both require busy waiting: while the process/thread waits, the CPU sits tight in a loop until the entry to the critical region is allowed. **Sleep** is a system call that causes the caller to block, that is, be suspended until another process wakes it up. The **wakeup** call has one parameter, the process to be awakened. Alternatively, both sleep and wakeup each have one parameter, a memory address used to match up sleeps with wakeups.

5. Semaphores

Dijkstra proposed using an integer variable to count the number of wakeups saved for future use. In his proposal, a new variable type, which he called a **semaphore**, was introduce. A semaphore could have the value 0, indicating that no wakeups were saved, or some positive value if one or more wakeups were pending. Semaphores have two operations: *down* and *up*. The *down* operation checks if the value is greater than 0; if so, it decrements the value (uses one stored wakeup) and just continues; if the value is 0, the process is put to sleep without completing the down for the moment. Checking the value, changing it, and possibly going to sleep, all are done as a single, indivisible **atomic action**. It is guaranteed that once a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked. This atomicity is absolutely essential to solving synchronaztion problems and avoiding race conditions. Atomic actions, in which a group of related operations are either all performed without interruption or not performed at all, are extremely important in many other areas of computer science as well. The *up* operation increments the value of the semaphore addressed. If one or more processes were sleeping on that semaphore, unable to complete an earlier down operation, one of them is chosen by the system, and is allowed to complete its down. Thus, after an up on a semaphore with processes sleeping on it, the semaphore will still be, there will be one fewer process sleeping on it. The operation of incrementing the semaphore and waking up one process is also indivisible. No process ever blocks doing an up, just as no process ever blocks doing a wakeup in the earlier model.

6. Mutexes

When the semaphore's ability to count is not needed, a simplified version of the semaphore, called a mutex, is sometimes used. Mutexes are good only for managing mutual exclusion to some shared resource or piece of code. They are easy and efficient to implement, which makes them especially useful in thread packages that are implemented entirely in user space. A **mutex** is a shared variable that can be in one of two states: unlocked or locked. Consequently, only 1

bit is required to represent it, but in pratice an integer often is used, with 0 meaning unlocked and all other values meaning locked. Two procedures are used with mutexes. When a thread (or process) needs access to a critical region, it calls mutex_lock. If the mutex is currently unlocked (meaning that the critical region is available), the call succeds and the calling thread is free to enter the critical region. On the other hand, if the mutex is already locked, the calling thread is blocked until the thread in the critical region is finished and calls mutex_unlock. If multiple threads are blocked on the mutex, one of them is chosen at random and allowed to acquire the lock. Because mutexes are so simple, they can easily be implemented in user space, provided that a TSL or XCHG instruction is available.

Pthreads

In addition to mutexes, Pthreads offer a second synchronazation mechanism: **condition variables**. Mutexes are good for allowing or blocking access to a critical region. Condition variables allow threads to block due to some condition not being met. Almost always the two methods are used together.

7. Monitors

When using semaphores, one must be careful; one subtle error and everything comes to a grinding halt. It is like programming in assembly language, only worse, because the errors are race conditions, deadlocks, and other forms of unpredictable and irreproducible behavior. To make it easier to write correct programs, Brinch Hanses and Hoare proposed a higher-level synchronization primitive called a **monitor**. Their proposal differed slightly. A monitor is a collection of processes, variables and data structures that are all grouped together in a special kind of module or package. Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor. C cannot be used here because monitores are a language concept and C does not have them. Monitors have an important property that makes them useful for achieving mutual exclusion: only one process can be active in a monitor at any instant. Monitors are a programming-language construct, so the compiler knows they are special and can handle calls to monitor procedures differently from other procedure calls. Typically, when a process calls a monitor procedure, the first few instructions of the procedure will check to see if any other process is currently active within the monitor. If so, the calling process wil be suspended until the other process has left the monitor. If no other process is using the monitor, the calling process may enter. It is up to the compiler to implement mutual exclusion on monitor entries, but a common way is to use a mutex or a binary semaphore. Because the compiler, not the programmer, is arranging for the mutual exclusion, it is much less likely that something will go wrong. In any event, the person writing the monitor does not have to be aware of how the

compiler arranges for mutual exclusion. It is sufficient to know that by turning all the critical regions into monitor procedures, no two processes will ever execute their critical regions at the same time. Although monitors provide an easy way to achieve mutual exclusion, as we have seen above, that is not enough. We also need a way for processes to block when they cannot proceed. The solution lies in **condition variables**, along with two operations on them, wait and signal. When a monitor procedure discovers that it cannot continue, it does a wait on some condition variable. This action causes the calling process to block. It also allows another process that had been previously prohibited from entering the monitor to enter now. This other process, can wake up its sleeping partner by doing a signal on the condition variable that its partner is waiting on. To avoid having two active processes in the monitor at the same time, we need a rule telling what happens after a signal. Hoare proposed letting the newly awakened process run, suspending the other one. Brinch Haisen proposed finessing the problem by requiring a process that a process doing a signal must exit the monitor immediately. In other words, a signal statement may appear only as the final statement in a monitor procedure. We will use the latter. If a signal is done on a condition variable on which several processes are waiting, only one of them, determined by the system scheduler, is revived. Condition variables are not counters. They do not accumulate signals for later use the way semaphores do. Thus, if a condition variable is signaled with no one waiting on it, the signal is lost forever. In other words, **the wait must come before the signal**. This rule makes the implementation much simpler. In practice, it is not a problem because it is easy to keep track of the state of each process with variables, if need be. A process that might otherwise do a signal can see that this operation is not necessary by looking at the variables. Nevertheless, monitors have their drawbacks. Most languages do not have monitors. Another problem is that they were designed for solving the mutual exclusion problem on or more CPUs that all have access to a common memory. By putting the semaphores in the shared memory and protecting them with TSL or XCHG instructions, we can avoid races. When we move to a distributed system consisting of multiple CPUs, each with its own private memory and connected by a local area network, these primitives become innapplicable. The conclusion is that semaphores are too low level and monitores are not usable except in a few programming languages. Also, none of the primites allow information exchange between machines. Something else is needed.

8. Message Passing

That something else is message passing. This method of interprocess communication uses two primitives, *send* and *receive*, which, like semaphores and unlike monitors, are system calls rather than language constructs. As such, they can easily be put into library procedures, such as:

- send (destination, &message);

- receive (source, &message); The former call sends a message to a given destination and the latter one receives a message from a given source (or from ANY, if the receiver does not care). If no message is available, the receiver can block until one arrives. Alternatively, it can return immediately with an error code.

Design Issues for Message-Passing Systems

Message-passing systems have many problems and design issues that do not arise with semaphores or with monitors, especially if the communicating processes are one different machines connected by a network. For example, messages can be lost by network. To guard against lost messages, the sender and receiver can agree that as soon as a message has been received, the receiver will send back a special **acknowledgement** message. If the sender has not received the acknowledgement within a certain time interval, it retransmits the message. Now consider what happens if the message is received correctly, but the acknowledgement back to the sender is lost. The sender will retransmit the message, so the receiver will get it twice. It is essential that the receiver be able to distinguish a new message from the retransmission of an old one. Usually, this problem is solved by putting consecutive sequence numbers in each original message. If the receiver gets a message bearing the same sequence number as the previous message, it knows that the message is a duplicate that can be ignored. Successfully communicating in the face of unreliable message passing is a major part of the study of computer networks. Message systms also have to deal with the question of how processes are named, so that the process specified in a send or receive call is unambiguous. **Authentication** is also an issue in message systems: how can the client tell that it is communicating with the real file server, and no with an imposter? At the other end of the spectrum, there are also design issues that are important when the sender and receiver are on the same machine. One of these is performance. Copying messages from one process to another is always slower than doing a semaphore operation or entering a monitor. Much work has gone into making message passage efficient.

Message passing is commonly used in parallel programming systems. One well-known message passing system, for example, is MPI (Message-Passing Interface).