

Revision Notes

A *concurrent program* is one consisting of two or more processes. Each process is a sequential program. In a concurrent program, there are multiple threads of execution or control.

Concurrent programs are often structured using the same programming techniques as sequential programs, but they are intrinsically more complex:

- when more than one activity can occur at a time, program execution is necessarily nondeterministic;
- code may execute in surprising orders - any order that is not explicitly ruled out is allowed, e.g., a field set to one value in one line of code may have a different value *before the next line of code is executed*.
- concurrent programming requires more rigour on the part of the programmer.

We can distinguish two main types of implementations of concurrency:

- *shared memory*: the execution of concurrent processes by running them on one or more processors all of which access a shared memory - processes communicate by reading and writing shared memory locations
- *distributed processing*: the execution of concurrent processes by running them on separate processors - processes communicate by message passing.

We can further distinguish between:

- *multiprogramming*: the execution of concurrent processes by timesharing them on a single processor (concurrency is *simulated*);
- *multiprocessing*: the execution of concurrent processes by running them on separate processors which all access a shared memory (*true parallelism* as in distributed processing).

It is often convenient to ignore this distinction when considering shared memory implementations.

Concurrency is useful both when we want a program to do several things at once, and as an implementation strategy. In real-time systems, concurrency is implicit in the *specification* of the problem whereas in parallel programming concurrency occurs at the level of the *implementation* where there may be no concurrency in the problem requirements, e.g., in scientific computations.

Java supports both shared memory and distributed processing implementations of concurrency. Using shared memory, there are multiple user threads in a single Java Virtual Machine where the threads communicate by reading and writing shared memory locations. In a distributed processing environment, threads in different JVMs communicate by message passing or remote procedure call.

A *process* is any thread of execution or control, e.g. part of a concurrent program or a program running in different address spaces on the same processor (OS processes). A Java *thread* is a single sequential flow of control within a program. Within the JVM, the *threads* comprising a Java program are represented by instances of the Thread class.

There are two ways to create a thread in Java:

- extending the Thread class and overriding its run() method
`class Particle extends Thread { ... }`
- defining a class which implements the Runnable interface and defining the run() method
`class ParticleCanvas extends Canvas implements Runnable { ... }`

Note: The Run method does not throw exceptions – must be caught within the method.

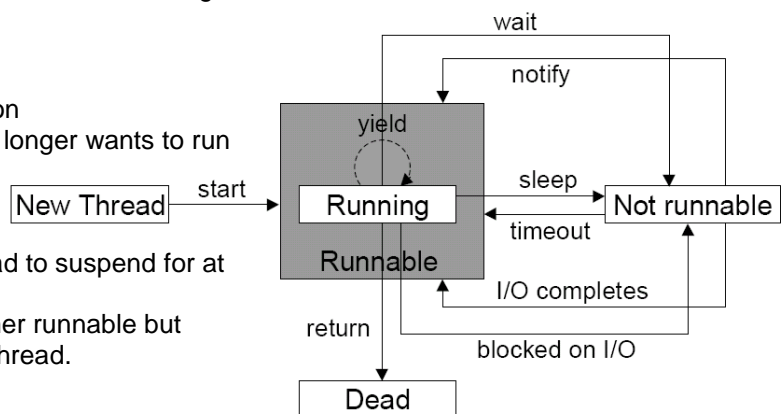
The Thread Lifecycle

A running Thread becomes not runnable when:

- it blocks in wait() for condition synchronisation
- it calls sleep() to tell the scheduler that it no longer wants to run
- it blocks for I/O

Scheduling methods

- sleep(long msec): causes the current thread to suspend for at least msec milliseconds.
- yield(): requests that the JVM to run any other runnable but non-running thread rather than the current thread.



Revision Notes

Thread priorities

Threads have *priorities* which heuristically influence schedulers:

- each thread has a priority in the range `Thread.MIN_PRIORITY` to `Thread.MAX_PRIORITY`
- by default, each new thread has the same priority as the thread that created it. The initial thread associated with a main method by default has priority `Thread.NORM_PRIORITY`
- the current priority of a thread can be accessed by the method `getPriority()` and set via the method `setPriority(int priority)`.
- Priorities are not set in stone – the scheduler may run any thread to prevent starvation.

When there are more runnable threads than CPUs, a scheduler is generally biased in favour of threads with higher priorities.

There are several ways to get a thread to stop:

- when the thread's `run()` method returns
- call `Thread.stop()` - this is a bad idea, as it doesn't allow the thread to clean up before it dies
- `interrupt()` the thread:
 - Each Thread object has an associated boolean interruption status. `interrupt()` sets the thread's interrupted status to *true*. `isInterrupted()` returns *true* if the thread has been interrupted by `interrupt()`.
 - A thread can periodically check its interrupted status, and if it is *true*, clean up and exit.

A multi-threaded program will continue to run until its last (non-daemon) thread terminates.

Synchronization

The concurrent processes which constitute a concurrent program must cooperate to get anything done, e.g., downloading a file in a web browser generally creates a new process to handle the download. While the file is downloading you can also continue to scroll the current page, or start another download, as this is managed by a different process: The two processes must co-operate, e.g., both processes may want to update the display. For the processes to co-operate, they must be able to synchronise their actions.

There are two main synchronisation problems in concurrent programming:

Mutual Exclusion: ensuring that statements in different processes cannot execute at the same time.

Condition Synchronisation: delaying a process until some Boolean condition is true. This is usually implemented by having one process wait for an event that is signalled by another process.

Example: a Shared Buffer

The producer writes data into the buffer and the consumer reads data from the buffer. Head is the index of the item at the head of the queue, and tail is the index of the last item in the queue.

Mutual exclusion is used to ensure that the producer and consumer do not access the same buffer slot at the same time, i.e., that partial data is not read or that partially read data is not overwritten; condition synchronisation is used to ensure that the consumer doesn't get too far ahead of the producer and vice versa, i.e., data is not read before it has been written, and that data is not overwritten before it has been read.

The problems of mutual exclusion and condition synchronisation also apply to functionally independent processes which don't co-operate, for example, separate programs on a time-shared computer as such programs implicitly compete for resources so they still need to synchronise their actions, e.g., two programs can't use the same printer at the same time or write to the same file at the same time. In this case, synchronisation is handled by the OS, using similar techniques to those found in concurrent programs.

Consider a multiprogramming implementation of a concurrent program consisting of two processes. The switching between processes occurs voluntarily (e.g., yield() in Java) or in response to interrupts, which signal external events such as the completion of an I/O operation or clock tick to the processor.

The processor executes a sequence of instructions which is an interleaving of the instruction sequences from each process. The process switching does not affect the order of instructions executed by each process.

Multiprocessing implementations of concurrency can be modelled in the same way: Each program statement or machine instruction ultimately reduces to a sequence of atomic actions, e.g., loading and storing registers. The effect of executing a set of atomic actions in parallel is equivalent to executing them in some arbitrary serial order, since the state transformations caused by an atomic action are indivisible, and hence cannot [by definition] be affected by atomic actions executed in parallel with it.

When modelling concurrency, we assume that concurrency is modelled as interleaving, processes execute at arbitrary relative speeds (a process can take arbitrarily long to proceed from one instruction to the next) and that instructions from processes are arbitrarily interleaved. This is referred to as an asynchronous model of execution.

Interference

If instructions from different processes are arbitrarily interleaved, any interleaving which is not explicitly prohibited is allowed. Interference occurs when two processes read and write shared variables in an unpredictable order, and hence with unpredictable results. Inevitably, some interleavings will have results you don't want:

Process 1 // initialisation code tail = tail + 1; queue[tail] = data1; // other code ...	Process 2 // initialisation code tail = tail + 1; queue[tail] = data2; // other code ...
Shared datastructures Object queue[SIZE]; integer tail;	

Consider the following execution sequence:

Let Tail = 6

P1: tail = tail + 1;

P2: tail = tail + 1;

P2: queue[tail] = data2;

P1: queue[tail] = data1;

This results in queue[7] being empty, and data2 being overwritten by data1 in queue[8].

Avoiding interference

To avoid interference, we need to ensure that no two processes access a shared variable at the same time. We do this by marking such sections of code as critical and requiring that no two processes are executing critical code at the same time. This process is called mutual exclusion.

A *critical section* is a section of code belonging to a process in a concurrent program that accesses a shared resource, e.g., a shared variable, shared communication channel, shared file etc.; and, for correct behaviour of the program, only one process may access the shared resource at a time. If processes A and B contain critical sections then the overlapped execution of process A and process B could result in interference. Mutual exclusion is the requirement that, at any given time, at most one process in a concurrent program is executing a critical section. Once one process has entered a critical section, no other process may enter a critical section until the first process has exited its critical section. Note that Mutual exclusion is a constraint on the execution of processes which applies between the process's critical sections, not between the processes themselves, i.e., the fact that A and B contain critical sections does not mean that their execution should never overlap, only that the execution of their critical sections should never overlap. For this to happen, one or more process may have to wait to enter its critical section, e.g., if process A is in its critical section, process B will have to wait to enter its critical section. In concurrent programs there are often a large number of critical sections which do not all need to be mutually exclusive with each other. A *class of critical sections* is a set of critical sections, all of which must be mutually exclusive with others in the same class. Critical sections from different classes do not need to be mutually exclusive.

Archetypical mutual exclusion

Any program consisting of n processes for which mutual exclusion is required between critical sections belonging to just one class can be written:

// Process 1 init1; while(true) { crit1; rem1; }	// Process 2 init2; while(true) { crit2; rem2; }	// Process n initn; while(true) { critn; remn; }
---	---	---

where $init_i$ denotes any (non-critical) initialisation, $crit_i$ denotes a critical section and rem_i denotes the (non-critical) remainder of the program, and i is 1, 2, ... n. We assume that $init$, $crit$ and rem may be of any size and that $crit$ must execute in a finite time but $init$ and rem may be infinite. $crit$ and rem may vary from one pass through the while loop to the next. With these assumptions it is possible to rewrite any process with critical sections into the archetypical form.

Mutual Exclusion and Atomic Actions

A process is the execution of a sequential program. The state of a process at any point in time consists of the values of both the program variables and some implicit variables, e.g., the program counter, contents of registers, etc. As a process executes, it transforms its state by executing statements, each of which consists of a sequence of one or more *atomic actions* that make indivisible state changes, i.e., uninterruptible machine instructions that load and store registers. Any intermediate state that might exist in the implementation of an atomic action is not visible to other processes.

Hardware assumptions:

- values of basic types are stored in memory locations, e.g., words, that are read and written as atomic actions
- values of program variables are manipulated by loading them into registers, modifying the register value and storing the results back into memory
- each process has its own set of registers, either:
 - real registers (in a multiprocessing) or
 - register values are saved and restored when switching processes (multiprogramming)
- when evaluating a complex expression, intermediate results are stored in registers or in memory private to the executing process, e.g., on a private stack.

Revision Notes

An *atomic action* is one that appears to take place as a single indivisible operation. A process switch can't happen during an atomic action, no other action can be interleaved with an atomic action and no other process can interfere with the manipulation of data by an atomic action.

There are two kinds of atomic action:

- a *fine-grained* atomic action is one that can be implemented directly as uninterruptible machine instructions e.g., loading and storing registers and
- a *coarse-grained* atomic action consists of an uninterruptible sequence of fine-grained atomic actions.

Reading and writing a *single* memory location are *atomic* operations. However, accesses to non-basic types, e.g. doubles, strings, arrays or reference types are (usually) not atomic and if data items are packed two or more to a word, e.g. strings and bitvectors, then write accesses may not be atomic. Few programming languages specify anything about the indivisibility of variable accesses, leaving this as an implementation issue.

Java is unusual in specifying which memory accesses are atomic:

- reads and writes to memory cells corresponding to instance or static fields and array elements of any type *except* long or double are guaranteed to be atomic
- when a non-long or double field is used in an expression, you will get either its initial value or some value that was written by some thread, however you are not guaranteed to get the value most recently written by any thread.

Many modern computers provide additional special indivisible instructions:

- Exchange instruction: $x \leftrightarrow r$
- Test-and-set instruction: `if (x == 0) { x = 1; r = 0; } else { r = 1; }`
- Increment & Decrement instructions: `x = x + 1; r = x;`
where x is a variable and r is a register.

Special machine instructions can be used to solve some simple mutual exclusion problems directly, e.g.:

- Single Word Readers and Writers
 - o Several processes read a shared variable and several process write to the shared variable, but no process *both reads and writes*. If the variable can be stored in a single word, then the memory unit will ensure mutual exclusion for all accesses to the variable. Also works in multiprocessing implementations.
- Shared Counter
 - o Several processes each increment a shared counter. If the counter can be stored in a single word, then a special *increment instruction* can be used to update the counter, ensuring mutual exclusion. Reading the value of the shared counter is also mutually exclusive.
 - o Doesn't work for multiprocessing implementations.

Multiprocessors

The normal indivisibility of many atomic actions does not provide mutual exclusion between different processors.

- disabling interrupts is local to one processor
- special machine instructions *do not provide mutual exclusion between different processors*, for example, test-and-set x is atomic on one processor, but a process on a different processor could modify the value of x during the execution of the test-and-set instruction.

Multiprocessor machines sometimes provide a special *memory lock* instruction which locks memory during execution of the *next* instruction.

- No other processors are permitted access to the shared memory during the execution of the instruction following the memory lock instruction.
- Memory locked instructions are thus effectively indivisible and therefore mutually exclusive across *all* processors.
- However memory locked instructions have to be used with care to avoid locking other processors, such as device controllers, out of memory for unacceptable periods of time.

Fine-grained atomic actions are not very useful to the applications programmer:

- atomic actions don't work for multiprocessor implementations of concurrency unless we can lock memory
- the set of atomic actions (special instructions) varies from machine to machine
- we can't assume that a compiler will generate a particular sequence of machine instructions from a given high-level statement
- the range of things you can do with a single machine instruction is limited - we can't write a critical section of more than one instruction.

Revision Notes

Coarse-grained atomic actions

To write critical sections of more than a single machine instruction, we need some way of concatenating fine-grained atomic actions. A *coarse-grained* atomic action consists of an uninterruptible sequence of fine-grained atomic actions, e.g., a call to a synchronized method in Java. They can be implemented at the hardware level by disabling interrupts, or by defining a mutual exclusion protocol.

Process switches happen between (fine-grained) atomic actions. In a multiprogramming implementation there are 3 points at which a process switch can happen:

- (hardware) interrupt, e.g., completion of an I/O operation, system clock etc.
- return from interrupt, e.g. after servicing an interrupt caused by a key press or mouse click
- trap instruction, e.g., a system call.

Disabling interrupts

We can ensure mutual exclusion between critical sections in a multiprogramming implementation by disabling interrupts while in a critical section. However this approach has several disadvantages:

- it is available only in privileged mode
- it excludes *all* other processes, reducing concurrency
- it doesn't work in multiprocessing implementations.

Disabling interrupts is only useful in a small number of situations, such as writing operating systems, dedicated systems or bare machines such as embedded systems or simple processors which don't provide support for multi-user systems. It is not a very useful approach from the point of view of an application programmer.

Defining a mutual exclusion protocol

Our protocol will consist of a sequence of instructions which is executed before (and possibly after) the critical section. Such protocols can be defined using standard sequential programming primitives, special instructions and what we know about when process switching can happen. Fine-grained atomic actions can be used to implement higher-level synchronisation primitives and protocols.

The synchronized keyword in Java can be used to define coarse-grained atomic actions:

- a synchronized method or block is executed under mutual exclusion with all other synchronized methods or blocks on the same object
- reads and write to a volatile field are guaranteed to be atomic, even for longs or doubles

Correctness of concurrent programs

A concurrent program must satisfy two types of property:

- Safety Properties: requirements that something should never happen, e.g., failure of mutual exclusion or condition synchronisation, deadlock etc.
- Liveness Properties: requirements that something will eventually happen, e.g. entering a critical section. Note that establishing liveness may require proving safety properties.

The protocols should satisfy the following properties:

- Mutual Exclusion: at most one process at a time is executing its critical section.
- Absence of Deadlock (Livelock): if two or more processes are attempting to enter their critical sections, at least one will succeed.
- Absence of Unnecessary Delay: if a process is trying to enter its critical section and other processes are executing their noncritical sections (or have terminated), the first process is not prevented from entering its critical section.
- Eventual Entry: a process that is attempting to enter its critical section will eventually succeed.

A solution using spin locks

```
bool lock = false;          // shared lock variable
initialisations;
while(true) {
    while(lock) { }; ← entry protocol
    lock = true;
    critical operations;
    lock = false;          ← exit protocol
    remainder of program;
}
```

Revision Notes

Does this solution satisfy the following properties:
Mutual Exclusion: no

Consider this sequences of events:

Process 1	Process 2
Enter inner while loop Check lock. Find it to be false.	Enter inner while loop Check lock. Find it to be false.
Set lock to be true Do critical operations	Set lock to be true Do critical operations

As is evident, there is a problem in that the lines while (lock) { }; and lock = true; must be executed atomically if the protocol is to work.

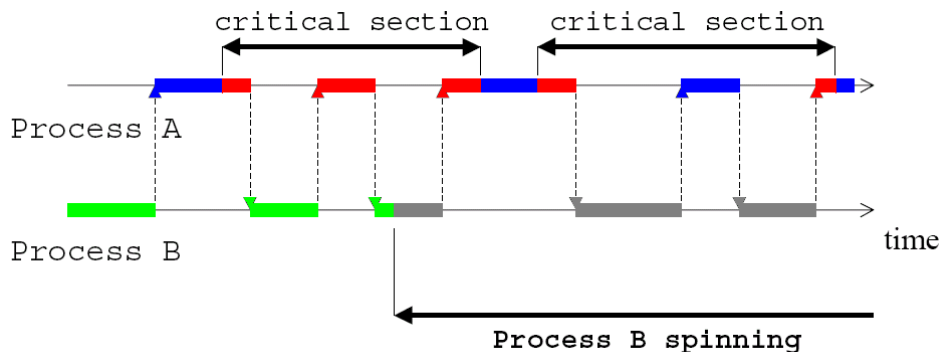
Solution: The Test-and-Set instruction effectively executes the function

```
bool TS(bool &lock) {
    bool v = lock;
    lock = true;
    return v;
} as an atomic action.
```

```
bool lock = false; // shared lock variable
// Process i
initi;
while(true) {
    while(TS(lock)) { }; // entry protocol
    criti;
    lock = false; // exit protocol
    remi;
}
```

Test-and-Set must be atomic. In a multiprocessing implementation, Test-and-Set must effectively lock memory. If the critical sections are short, the amount of time that each process should have to spend busy waiting will be small. If both processes don't try to enter their critical section at the same time neither will have to wait. Since all processes execute the same protocol it works for any number of processes.

This protocol fulfills all the properties needed, with the possible exception of the Eventual Entry criterion which is only guaranteed if the scheduling policy is strongly fair. Otherwise starvation may occur:



Dekkers's Algorithm

integer c1 = c2 = turn = 1; // shared variables	
<pre>// Process 1 init1; while(true) { c1 = 0; // entry protocol while (c2 == 0) { if (turn == 2) { c1 = 1; while (turn == 2) {}; c1 = 0; } crit1; turn = 2; // exit protocol c1 = 1; rem1; }</pre>	<pre>// Process 2 init2; while(true) { c2 = 0; // entry protocol while (c1 == 0) { if (turn == 1) { c2 = 1; while (turn == 1) {}; c2 = 0; } crit2; turn = 1; // exit protocol c2 = 1; rem2; }</pre>

What happens if Process 1 is slow setting c1 to 1? Process 2 just spins in the outer loop for a little longer. As turn has been set to 2, the if statements are not executed, i.e. it doesn't back off.

What happens if Process 1 and Process 2 simultaneously try to enter their critical sections? The turn variable will ensure only Process 1 gets control of the critical variables.

What would happen if we swapped the order of the statements in the exit protocol? Is the algorithm still correct? It is still correct.

The solution based on Dekker's algorithm has the all the desirable properties. Eventual Entry is guaranteed even if scheduling policy is only *weakly fair*.

	Test-and-Set	Dekker's Algorithm
Mutual Exclusion:	yes	yes
Absence of Deadlock:	yes	yes
Absence of Unnecessary Delay:	yes	yes
Eventual Entry:	scheduling strongly fair	scheduling weakly fair
Practical issues:	special instructions, any number of processes	standard instructions, > 2 processes complex

Java

Java code optimisation

Not only may concurrent executions be interleaved, they may also be reordered and otherwise manipulated to increase execution speed:

- the compiler may rearrange the order of the statements
- the processor may rearrange the execution order of the machine instructions
- the memory system may rearrange the order in which writes are committed to memory
- the compiler, processor and/or memory system may maintain variable values in, e.g., CPU registers, rather than writing them to memory so long as the code has the "intended" effect.

Java allows threads that access shared variables to keep private working copies of variables:

- each thread is defined to have a *working memory* (an abstraction of caches and registers) in which to store values;
- this allows a more efficient implementation of multiple threads.

Revision Notes

The Java Memory Model specifies when values must be transferred between main memory and per-thread *working memory*:

- **Atomicity:** which instructions must have indivisible effects
Reads and writes to memory cells corresponding to fields of any type *except* long or double are guaranteed to be atomic. When a field (other than long or double) is used in an expression, you will get either its initial value or some value that was written by some thread - not necessarily the value most recently written by any thread.
- **Visibility:** under what conditions are the effects of one thread visible to another
Without synchronization, changes to fields made by one thread are not guaranteed to be visible to other threads. The first time a thread accesses a field of an object, it sees either the initial value of the field or a value since written by some other thread. When a thread terminates, all written variables are flushed to main memory.
- **Ordering:** under what conditions the effects of operations can appear out of order to any given thread.
The apparent order in which the instructions in a method are executed can differ. From the point of view of the thread executing the method, instructions *appear* to be executed in the proper order (*as-if-serial* semantics). From the point of view of other threads executing unsynchronised methods almost anything can happen.

Synchronisation

When synchronisation is used consistently, each of these properties has a simple characterisation:

- all changes made in one synchronized method or block are atomic and visible with respect to other synchronized methods or blocks employing the same lock
- processing of synchronized methods or blocks within any given thread is in program-specified order.

Volatile fields

If a field is declared volatile, a thread must reconcile its working copy of the field with the master copy every time it accesses the variable.

- Reads and writes to a volatile field are guaranteed to be atomic (even for longs and doubles)
 - New values are immediately propagated to other threads
 - From the point of view of other threads, the relative ordering of operations on volatile fields are preserved.
- However the ordering and visibility effects surround only the single read or write to the volatile field itself, e.g. '++' on a volatile field is not atomic.

Dekker's algorithm implicitly relies on variable reads and writes being atomic, the values written to the variables being immediately propagated to the other process (thread) and the ordering of the instructions being preserved.

Spin locks in Java

With unsynchronized code, all that is guaranteed by the Java Memory Model is that the variable reads and writes are atomic. We may have to wait an arbitrarily long time for new values of, e.g., `c1` or `turn`, to be propagated to the other thread and an optimising compiler could reorder the instructions in the threads, so long as the threads themselves can't tell the difference, e.g., the compiler could swap the order of `c1 = 1` and `while (turn == 2) { }` inside the outer loop, since the thread can't tell the difference.

Threads have priorities which heuristically influence schedulers. Each thread has a priority in the range `Thread.MIN_PRIORITY` to `Thread.MAX_PRIORITY`. When there are more runnable threads than CPUs, a scheduler is generally biased in favour of threads with higher priorities.

Typically, a thread will run until one of the following conditions is true:

- a higher-priority thread becomes runnable
- the thread yields or its `run()` method exits
- on systems that support time-slicing, its quantum has expired.

In general, lower-priority threads will run only when higher-priority threads are blocked (not runnable).

When considering archetypical mutual exclusion, we assumed that the initialisation, critical sections and remainder may be of any size any may take any length of time to execute and that each may vary from one pass through the while loop to the next. Also that the critical sections must execute in a finite time, i.e., each process must leave its critical section after a finite period of time and that the initialisation and remainder of each process may be infinite. However Java makes *no* promises about scheduling or fairness, and does not even strictly guarantee that threads make forward progress. Most Java implementations display some sort of weak, restricted or probabilistic fairness properties with respect to executing runnable threads, however you can't depend on this.



Revision Notes

A consequence of Java's weak scheduling guarantees is that spin locks of the form `while (turn == 2) {}` may spin *forever*. Even a loop of the form `while (turn == 2) Thread.yield();` is not guaranteed to be effective in allowing other threads to execute and change the condition.

Problems with Dekker's algorithm

Dekker's algorithm is *correct*, however it is complex and inefficient. Solutions to the Mutual Exclusion problem for n processes are quite complex, especially if we want to ensure the Eventual Entry property. It uses busy-waiting (spin locks) to achieve synchronisation, which is often unacceptable in a multiprogramming environment when you consider the overhead of spin locks: Time spent spinning is *wasted CPU* - Process B can do no useful work while Process A is in its critical section. However, the scheduler doesn't know this, and will (repeatedly) try to run Process B even while process A is in its critical section. If the critical sections are large relative to the rest of the program, or there are a large number of processes contending for access to the critical section, this will slow down your concurrent program, e.g., with 10 processes competing to access their critical sections, in the worst case we could end up wasting 90% (or more) of the CPU.

Semaphores

A *semaphore* s is an integer variable which can take only non-negative values. Once it has been given its initial value, the only permissible operations on s are the atomic actions:

- $P(s)$: if $s > 0$ then $s--$, else suspend execution of the process that called $P(s)$
- $V(s)$: if some process p is suspended by a previous $P(s)$ on this semaphore then resume p , else $s++$

A *general semaphore* can have any non-negative value; a *binary semaphore* is one whose value is always 0 or 1.

A semaphore can be seen as an abstract data type, having a set of permissible values and a set of permissible operations on instances of the type. However, unlike normal abstract data types, we require that the P and V operations on semaphores be implemented as *atomic actions*.

Implementing suspension

There are several ways a processes can be suspended:

- busy waiting - this is inefficient
- blocking: a process is *blocked* if it is waiting for an event to occur without using any processor cycles.

Note that the definition of V doesn't specify which process is woken up if more than one process has been suspended on the same semaphore.

We can think if P and V as controlling *access to a resource*:

- When a process wants to use the resource, it performs a P operation. If this succeeds, it decrements the amount of resource available and the process continues.
- If all the resource is currently in use, the process has to wait.
- When a process is finished with the resource, it performs a V operation. If there were processes waiting on the resource, one of these is woken up. If there were no waiting processes, the semaphore is incremented indicating that there is now more of the resource free.

```
binary semaphore s = 1; // shared binary semaphore
// Process i
inti;
while(true) {
    P(s); // entry protocol
    criti;
    V(s); // exit protocol
    remi;
}
```

The semaphore solution satisfies the four required properties, although *Eventual Entry* is only guaranteed if the semaphores are *fair*.

The semaphore solution works for n processes, it is much simpler than an n process solution based on Dekker's algorithm and it avoids busy waiting.

If we have n processes, of which k can be in their critical section at the same time, we can use a general semaphore, s initialised to k

Semaphores and condition synchronisation

Condition synchronisation involves delaying a process until some boolean condition is true. Such problems can be solved using *busy waiting* - the process simply sits in a loop until the condition is true. This is inefficient. Semaphores are not only useful for implementing mutual exclusion, but can be used for general condition synchronisation:

Example: Producer-Consumer with an infinite buffer

Given two processes, a *producer* which generates data items, and a *consumer* which consumes them, we assume that the processes communicate via an infinite shared buffer. The producer may produce a new item at any time and the consumer may only consume an item when the buffer is not empty. All items produced are eventually consumed.

<pre>// Shared variables Object[] buf = new Object[size]; semaphore n = 0;</pre>	
<pre>// Producer process Object v = null; integer in = 0; while(true) { // produce data v buf[in] = v; in++; V(n); }</pre>	<pre>// Consumer process Object w = null; integer out = 0; while(true) { P(n); w = buf[out]; out++; // use the data w }</pre>

Implementing semaphores

The semaphore operations **P** and **V** can be implemented using any of the techniques we have seen for ensuring mutual exclusion we have seen:

- mutual exclusion algorithms (e.g., Dekker's algorithm)
- special hardware instructions (e.g. Test-and-Set)
- disabling interrupts

The Producer-Consumer problem

Given two processes, a producer which generates data items, and a consumer which consumes them, find a mechanism for passing data from the producer to the consumer such that:

- no items are lost or duplicated in transit
- items are consumed in the order they are produced
- all items produced are eventually consumed

The single Producer–single Consumer problem can be generalized to multiple producers–single consumer, single producer–multiple consumers and multiple producers–multiple consumers.

In multiprogramming or multiprocessing implementations of concurrency, communication between a producer and a consumer is often implemented using a shared buffer. A buffer is an area of memory used for the temporary storage of data while in transit from one process to another. The producer writes into the buffer and the consumer reads from the buffer, e.g., a Unix pipe.

The general multiple Producer-multiple Consumer problem requires both mutual exclusion and condition synchronisation:

- mutual exclusion is used to ensure that more than one producer or consumer do not access the buffer at the same time.
- condition synchronisation is used to ensure that a message is not read before it has been written, and that a message is not overwritten before it has been read.

Synchronisation can be achieved using any of the techniques we have seen so far: e.g., Dekker's algorithm, semaphores.

Buffer-based solutions to the Producer–Consumer problem should satisfy the following conditions:

- no "items" are read from an empty buffer
- data items are read only once
- data items are not overwritten before they are read
- items are consumed in the order they are produced and
- all items produced are eventually consumed.

in addition to the properties of Mutual Exclusion, Absence of Deadlock, Absence of Unnecessary Delay and Eventual Entry.

Revision Notes

A Single element buffer

- The producer may only produce an item when the buffer is empty
- The consumer may only consume an item when the buffer is full.

The solution should satisfy the general synchronisation requirements for the Producer-Consumer problem and the properties of Mutual Exclusion, Absence of Deadlock, Absence of Unnecessary Delay and Eventual Entry.

With an infinite buffer we had only one problem - to prevent the consumer getting ahead of the producer. With a single element buffer we have two problems - preventing the consumer getting ahead of the producer *and* preventing the producer getting ahead of the consumer.

Simply surrounding the buffer read/writes with P(s) and V(s) where s is a binary semaphore initialized to 1 does not satisfy the synchronisation conditions:

- Items could be read from an empty buffer – the semaphore is initialised to 1 (otherwise when the producer comes to produce, it would have to wait) so if the consumer go there first, it would go ahead to successfully perform a P operation and read.
- Data items could be read multiple times – consider two successive reads with no write between. The first read performs a V enabling the second read to go ahead.
- Data items could be overwritten before they are read – same as above but for writers.
- All items produced are not always eventually consumed: consequence of above.

Items *are* consumed in the order they are produced.

Using the same method as with the infinite buffer does not give Mutual Exclusion (both processes are accessing the buffer whereas before, each process kept a local pointer – in and out). It solves the problem of data items being read from an empty buffer.

Data items are guaranteed to be read at most once, but maybe not at all: When an item is read, the V operation means another read cannot take place until another item is produced. However, there is nothing to stop two successive productions, the second overwriting the first.

Items *are* consumed in the order they are produced.

Solution

// Shared variables Object buf; Binary semaphore empty = 1, full = 0;	
// Producer process Object x = null; while(true) { // produce data x ... P(empty); buf = x; V(full); }	// Consumer process Object y = null; while(true) { P(full); y = buf; V(empty); // use the data y ... }

A single element buffer works well if the Producer and Consumer processes run at the same rate:

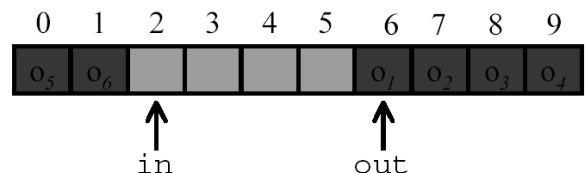
- processes don't have to wait very long to access the single buffer
- many low-level synchronisation problems are solved in this way, e.g., interrupt driven I/O.

If the speed of the Producer and Consumer is only the same on average, and fluctuates over short periods, a larger buffer can significantly increase performance by reducing the number of times processes block.

Bounded buffers

A bounded buffer of length n is a circular communication buffer containing n slots. The buffer contains a queue of items which have produced but not yet consumed.

- The producer may only produce an item when there is an empty slot in the buffer and
- The consumer may only consume an item when there is a full slot in the buffer.



Revision Notes

Note that the synchronisation conditions are really the same as for the single element (& infinite) buffer and the problems are the same (preventing the consumer getting ahead of the producer and preventing the producer getting ahead of the consumer). Hence, we can simply adapt the solution given for the single element buffer to use a general, instead of a binary, semaphore. Note that the in and out counters are local to each process and that n – the size of the buffer – is only ever read so the counter inc/decrements do not have to be in the critical section.

<pre>// Shared variables final integer n = buffersize; Object[] buf = new Object[n]; general semaphore empty = n, full = 0;</pre>	
<pre>// Producer process Object x = null; integer in = 0; while(true) { // produce data x ... P(empty); buf[in] = x; V(full); in = (in + 1) % n; }</pre>	<pre>// Consumer process Object y = null; integer out = 0; while(true) { P(full); y = buf[out]; V(empty); out = (out + 1) % n; // use the data y ... }</pre>

Bounded buffers are used for serial input and output streams in many operating systems:

- Unix maintains queues of characters for I/O on all serial character devices such as keyboards, screens and printers.
- Unix pipes are implemented using bounded buffers.

Semaphores can be used to solve simple mutual exclusion and condition synchronisation problems. However,

- they are only *low-level*. Omitting a single V operation is likely to lead to deadlock and omitting a P operation may lead to a violation of mutual exclusion.
- they are *unstructured*: synchronisation code is typically dispersed throughout the whole program, rather than localised in well-defined regions.
- they *confuse conceptually distinct operations*: the same primitives are used to implement mutual exclusion and condition synchronisation.

Monitors

A *monitor* is an abstract data type representing a shared resource. We saw how semaphores can be used to *control access* to a shared resource - here we go further and *encapsulate* the shared resource: a monitor implements a shared data structure (resource) together with the operations which manipulate the data structure.

Monitors have four components:

- a set of *private variables* which represent the state of the resource
- a set of *monitor procedures* which provide the public interface to the resource
- a set of *condition variables* used to implement condition synchronisation
- *initialisation code* which initialises the private variables.

Monitor procedures manipulate the values of the private monitor variables. Only the names of monitor procedures are visible outside the monitor. The only way a process can read or change the value of a private monitor variable is by calling a monitor procedure. The private monitor variables are shared by all the monitor procedures. Monitor procedures may also have their own local variables – each procedure call gets its own copy of these. Statements within monitor procedures (or initialisation code) may not access variables declared outside the monitor.

Condition variables are used to delay a process that can't safely execute a monitor procedure until the monitor's state satisfies some boolean condition. Condition variables are not visible outside the monitor and the only access to them is via *special monitor operations* within monitor procedures. Like semaphores, their values can't be tested or assigned to directly even by the monitor procedures.

Synchronisation within monitors is achieved using monitor procedures and condition variables. Mutual exclusion is implicit - monitor procedures by definition execute with mutual exclusion. Condition synchronisation must be programmed explicitly using *condition variables*.

At most *one* instance of *one* monitor procedure may be active in a monitor at a time. If one process is executing a monitor procedure and another process calls a procedure of the same monitor, the second process blocks and is placed on the *entry queue* for the monitor. When the process in the monitor completes its monitor procedure call, mutual exclusion is passed to the first blocked process on the entry queue. Entry queues are usually defined to be FIFO, so the first process to block will be the next to one to enter the monitor.

If all communication between processes is via calls to monitor procedures and the monitors contain all the shared state of the system in their private variables, then any accesses to any part of the shared state by any process is guaranteed to be mutually exclusive of any other accesses to that part of the state by other processes.

The value of a condition variable is a *delay queue* of blocked processes waiting on a condition. If a call to a monitor procedure can't proceed until the monitor's state satisfies some boolean condition, the process that called the monitor procedure *waits* on the corresponding condition variable. When another process executes a monitor procedure that makes the condition true, it *signals* to the process(es) waiting on the condition variable. Condition variables are like semaphores used for condition synchronisation.

We assume that the following operations are defined for a condition variable *v*:

- **wait(v)**: wait at the end of the delay queue for *v*
If a process can't proceed, it blocks on a condition variable *v* by executing **wait(v)**. The blocked process relinquishes exclusive access to the monitor and is appended to the end of the delay queue for *v*.
- **signal(v)**: wake the process at the front of the delay queue for *v* and continue
Processes blocked on a condition variable *v* are woken up when some *other* process performs **signal(v)**. This awakens the process at the front of the delay queue. If the delay queue for *v* is empty, **signal** does nothing. This is *unlike* semaphores, where if no process was waiting on the semaphore, a **V** operation increments the semaphore.
- **signal_all(v)**: wake all the processes on the delay queue for *v* and continue
- **empty(v)**: true if the delay queue for *v* is empty

Revision Notes

Signalling disciplines

When a monitor procedure calls **signal** on a condition variable, it wakes up the first blocked process in the delay queue waiting on the condition.

- *Signal and Wait*: the signaller waits until some later time and the signalled process executes now.
- *Signal and Continue*: the signaller continues and the signalled process executes at some later time.

The examples in this lecture use the *signal and continue* signalling discipline.

Example: Bounded Buffer with monitors

```
monitor BoundedBuffer {
    // Private variables ...
    Object buf = new Object[n];
    integer out = 0,           // index of first full slot
    in = 0,                   // index of first empty slot
    count = 0;               // number of full slots

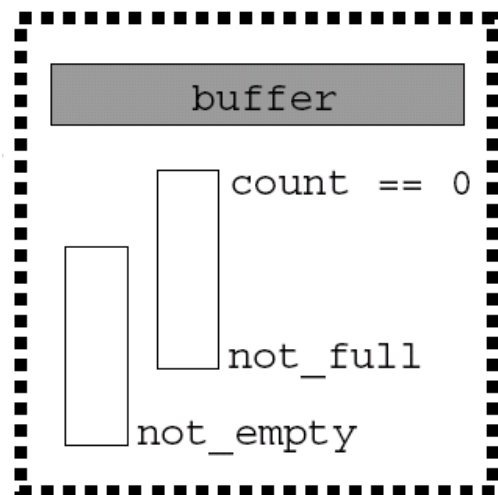
    // Condition variables ...
    condvar not_full,        // signalled when count < n
    not_empty;              // signalled when count > 0

    // Monitor procedures ...
    //(signal & continue signalling discipline)

    procedure append(Object data) {
        while(count == n) wait(not_full);
        buf[in] = data;
        in = (in + 1) % n;
        count++;
        signal(not_empty);
    }

    procedure remove(Object &item) {
        while(count == 0) wait(not_empty);
        item = buf[out];
        out = (out + 1) % n;
        count--;
        signal(not_full);
    }
}
```

BoundedBuffer Monitor



Semaphores and monitors have the same expressive power, i.e. monitors can be used to simulate semaphores and semaphores can be used to simulate monitors. What we gain with monitors is a higher level of abstraction.

Monitors & Java

Mutual exclusion can be implemented in Java using the `synchronized` keyword. A synchronized method (or block) is executed under mutual exclusion with all other synchronized methods on the same object. Java provides the basic operations for condition synchronisation: `wait()`, `notify()`, `notifyAll()`. Each object has a single (implicit) condition variable and delay queue, the *wait set* and uses the *signal and continue* signalling discipline.

Revision Notes

Readers and Writers

Given a shared file and a collection of processes that need to access and update the file:

- *reader* processes only need to read the file
- *writer* processes need to both read and write the file

We assume that the file is initially in a consistent state and that each read or write executed in isolation transforms the file into a new consistent state.

The Readers and Writers problem is an abstraction of a common class of concurrency problems:

The '*file*' can be any shared resource, e.g., an area of memory, or a database; '*reading*' and '*writing*' can be any operations. *Reading* doesn't change the resource - readers can be allowed to proceed concurrently. *Writing* must be mutually exclusive of all readers and all other writers.

Example: A simple library catalogue has two kinds of users - *borrowers* who use the catalogue to search for books or to find out whether a particular book is on loan (readers) and *library staff* who update the catalogue when new books are added to the library stock or to record which books are on loan (writers).

If we allow all users unrestricted access to the catalogue, a borrower may see the catalogue when it is an inconsistent state, e.g.:

- a borrower is looking for a book which has just arrived at the library
- a librarian is updating the catalogue to include the book, e.g., by copying and editing an existing entry
- the borrower sees an inconsistent state of the catalogue, e.g., the entry contains an incorrect shelf number.

The Readers and Writers problem is a special case of the general problem of a number of processes all of which may *both* read and write to a file. Any solution to the general problem will also be a *correct* solution to the Readers and Writers problem. However (much) more *efficient* solutions are possible for the Readers and Writers problem. In general, an efficient solution depends on the specifics of the problem.

One solution to the readers and writers problem would be to make all accesses to the file mutually exclusive:

- at most one process, whether reader or writer would be able to access the file at a time.
- this is simple and correct but, in general, the performance of such an approach is unacceptable, e.g., in the case of the library catalogue it would mean that only one reader could search the catalogue at a time.

To ensure correctness and for maximum efficiency (concurrency) and we require that a) if a writer is writing to the file, no other writer may write to the file and no reader may read it and b) any number of readers may simultaneously read the file.

Given a sequence of read and write requests which arrive in the order $R1 R2 W1 R3\dots$, in a *Readers' Preference* protocol: $R3$ takes priority over $W1$: $R1 R2 R3 W1\dots$

Given a sequence of read and write requests which arrive in the order $W1 W2 R1 W3\dots$, in a *Writers' Preference* protocol: $W3$ takes priority over $R1$: $W1 W2 W3 R1\dots$

A *fair* solution to the Reader's and Writer's problem:

- if there are waiting writers then a new reader is required to wait for the termination of a write
- if there are readers waiting for the termination of a write, they have priority over the next write.

As a (selective) *mutual exclusion* problem - classes of processes compete for access to the file. Reader processes compete with writer and individual writer processes compete readers and with each other.

As a *condition synchronisation* problem, reader processes must wait until no writers are accessing the file and writer processes must wait until there are no readers or other writers accessing the file.

Although the file is shared, we can't encapsulate it in a monitor, since the readers couldn't then access it concurrently. Instead the monitor is used to *arbitrate* access to the file - the file itself is global to the readers and writers. This basic structure is often employed in monitor based programs.

The arbitration monitor grants permission to access the file. Processes inform the monitor when they want access to the file (access requests) and when they are finished (release requests). With two kinds of access request (read and write) and two release requests (read and write), the monitor has four procedures: `startRead()`, `endRead()`, `startWrite()` and `endWrite()`.

```
monitor Solution {
```

Revision Notes

```
boolean writing = false;
integer readers = 0,
waitingWriters = 0;
condvar okToRead, okToWrite;

procedure startRead() {
    while (writing or waitingWriters > 0) wait(okToRead);
    readers++;
}

procedure endRead() {
    readers--;
    if (readers == 0) signal(okToWrite);
}

procedure startWrite() {
    if (writing or readers > 0 or waitingWriters > 0) {
        waitingWriters++;
        wait(okToWrite);
        waitingWriters--;
    }
    writing = true;
}

procedure endWrite() {
    writing = false;
    if (waitingWriters > 0) signal(okToWrite);
    else signal_all(okToRead);
}
}
```

Revision Notes

```
monitor ReadersPreference {

    boolean writing = false;
    integer readers = 0,
    waitingReaders = 0;
    condvar okToRead, okToWrite;

    procedure startRead() {
        if (writing) {                               // note the lack of `or
            waitingWriters > 0'
                waitingReaders++;
                wait(okToRead);
                waitingReaders--;
            }
        readers++;
    }

    procedure endRead() {
        readers--;
        if (readers == 0) signal(okToWrite);
    }

    procedure startWrite() {
        while (writing or readers > 0 or waitingReaders > 0) wait(okToWrite);
        writing = true;
    }

    procedure endWrite() {
        writing = false;
        if (waitingReaders > 0) signal_all(okToRead);
        else signal(okToWrite);
    }
}
```

```
monitor WritersPreference {

    boolean writing = false;
    integer readers = 0,
    waitingWriters = 0;
    condvar okToRead, okToWrite;

    procedure startRead() {
        while (writing or waitingWriters > 0) wait(okToRead);
        readers++;
    }

    procedure endRead() {
        readers--;
        if (readers == 0) signal(okToWrite);
    }

    procedure startWrite() {
        if (writing or readers > 0 or waitingWriters > 0) {
            waitingWriters++;
            wait(okToWrite);
            waitingWriters--;
        }
        writing = true;
    }

    procedure endWrite() {
        writing = false;
        if (waitingWriters > 0) signal(okToWrite);
        else signal_all(okToRead);
    }
}
```

Revision Notes

```
monitor FairSolution {  
  
    boolean writing = false;  
    integer readers = 0,  
    waitingReaders = 0,  
    waitingWriters = 0;  
    condvar okToRead, okToWrite;  
  
    procedure startRead() {  
        if (writing or waitingWriters > 0) {  
            waitingReaders++;  
            wait(okToRead);  
            waitingReaders--;  
        }  
        readers++;  
    }  
  
    procedure endRead() {  
        readers--;  
        if (readers == 0) signal(okToWrite);  
    }  
  
    procedure startWrite() {  
        if (writing or readers > 0 or waitingReaders > 0 or waitingWriters > 0) {  
            waitingWriters++;  
            wait(okToWrite);  
            waitingWriters--;  
        }  
        writing = true;  
    }  
  
    procedure endWrite() {  
        writing = false;  
        if (waitingReaders > 0) signal_all(okToRead);  
        else signal(okToWrite);  
    }  
}
```

In practice, exclusive writes and concurrent reads are not sufficient to gain the necessary performance in large database systems. The internal structure of the database must be used to limit the area to which a write lock is imposed then this area then behaves as if it were supporting a readers and writers protocol.

Synchronisation in Java

Java provides built-in support for mutual exclusion with the `synchronized` keyword. Changes made in one synchronized method (or block) are *atomic* and *visible* with respect to other synchronized methods (blocks) on the same object. There is a lock for each object - when a synchronized method (or block) is called, it waits to obtain the lock, executes the body of the method (block) and then releases the lock. Reads and writes to `volatile` fields are *atomic* and *visible* to other threads.

Java provides built-in support for condition synchronisation with the methods `wait()`, `notify()` and `notifyAll()`:

- to delay a thread until some condition is true, write a `wait()` loop that causes the thread to block if the delay condition is false
- ensure that every method which changes the truth value of the delay condition notifies threads waiting on the condition (using `notify()` or `notifyAll()`), causing them to wake up and re-check the delay condition.

`wait()`, `notify()` and `notifyAll()` must be executed within synchronized methods or blocks. `wait()` releases the lock on the object held by the calling thread—the thread blocks and is added to the *wait set* for the object. `notify()` wakes up *a thread* in the wait set (if any) and `notifyAll()` wakes up *all threads* in the wait set (if any). The thread that invoked `notify()` / `notifyAll()` continues to hold the object's lock. The awakened thread(s) remain blocked and execute at some future time when they can reacquire the lock.

notify() vs notifyAll()

`notify()` can be used to increase performance when only *one* thread needs to be woken, i.e. when

- all threads in the wait set are waiting on the *same* delay condition and
- each notification enables at most one thread to continue and
- the possibility of an `interrupt()` during `notify()` is handled

`notifyAll()` is required when:

- the threads in the wait set are waiting on different conditions or
- a notification can satisfy multiple waiting threads

Each Thread object has an associated boolean interruption status. `interrupt()` sets the thread's interrupted status to *true*, `isInterrupted()` returns *true* if the thread has been interrupted by `interrupt()`. A thread can periodically check its interrupted status, and if it is *true*, clean up and exit.

Threads which are blocked by calls to `wait()`, `sleep()` and `join()` aren't runnable, and thus can't check the value of the interrupted flag. Interrupting a thread which is not runnable aborts the thread, throws an `InterruptedException` and sets the thread's interrupted status to *false*. Therefore, calls to `wait()`, `sleep()`, or `join()` are often enclosed in a try catch block.

Mutual Exclusion

Mutual exclusion algorithms: pre- and post-protocols are implemented using special machine instructions or atomic memory accesses (e.g., Test-and-Set, Dekker's algorithm)

Semaphores: pre- and post-protocols can be implemented using atomic *P* and *V* operations

Monitors: mutual exclusion is *implicit* - pre- and post-protocols are executed automatically on entering and leaving the monitor to ensure that monitor procedures are not executed concurrently.

We can program any of these approaches in Java. It is possible to implement Dekker's algorithm in Java, though we must be careful to ensure that field values are propagated between threads. We can implement semaphores as a Java class with methods which implement the *P* and *V* operations. In fact, monitors are the basis of Java's synchronisation primitives - there is a straightforward mapping from designs based on monitors to solutions using synchronized classes.

Condition synchronisation

Busy-waiting: the process sits in a loop until the condition is true

Semaphores: *P* and *V* operations can be used to wait for a condition and to signal that it has occurred

Monitors: condition synchronisation is explicitly programmed using *condition variables* and monitor operations, e.g., `wait` and `signal`.

We can program any of these approaches in Java. It is possible to implement busy waiting in Java, though problems of visibility, scheduling and efficiency mean that using `wait()` and `notify()` is nearly always a better choice. We can implement semaphores as a Java class with methods which implement the *P* and *V* operations. While

Revision Notes

monitors are the basis of Java's synchronisation primitives, each object in Java has a only a single condition variable and delay queue - monitor operations can be implemented using wait() and notify().

```
class GeneralSemaphore {           // a semaphore example

    protected long resource;

    public GeneralSemaphore (long r) {
        resource = r;
    }

    public synchronized void P() throws InterruptedException {
        try {
            while (resource <= 0) wait();
            --resource;
        } catch (InterruptedException ie) {
            notify();
            throw ie;
        }
    }

    public synchronized void V() {
        ++resource;
        notify();
    }
}
```

```
class BoundedBuffer {           // a monitor example

    // Private variables ...
    Object[] buf;
    int out = 0,           // index of first full slot
    int in = 0,           // index of first empty slot
    int count = 0;       // number of full slots

    public BoundedBuffer(int n) {
        buf = new Object[n];
    }

    // Monitor procedures ...
    public synchronized void append(Object data) {
        try {
            while(count == n) wait();
        }
        catch (InterruptedException e) {
            return;
        }
        buf[in] = data;
        in = (in + 1) % n;
        count++;
        notifyAll();
    }

    public synchronized Object remove() {
        try {
            while(count == 0) wait();
        }
        catch (InterruptedException e) { return null; }

        Object item = buf[out];
        out = (out + 1) % n;
        count--;
        notifyAll();
        return item;
    }
}
```

Revision Notes

The safest design strategy based on mutual exclusion is to use *fully synchronized objects* in which all methods are synchronized, there are no public fields or other encapsulation violations, all methods are finite, all fields are initialised to a consistent state in constructors and the state of the object is consistent at both the beginning and end of each method (even in the presence of exceptions).

Synchronized methods and blocks have a number of limitations:

- there is no way to back off from an attempt to acquire a lock, e.g., to timeout or cancel a lock attempt following an interrupt.
Example: The `GeneralSemaphore` class is fully synchronized. When the `P()` method is invoked on an instance of the `GeneralSemaphore` class, `s`, the invoking thread attempts to obtain the lock on `s`. There is no way to back off if the lock is already held by another thread, to give up after waiting for a specified time, or to cancel the lock attempt following an interrupt. This can make it difficult to recover from liveness problems.
- synchronisation within methods and blocks limits use to strict block structured locking
- there is no way to alter the semantics of a lock, e.g., read vs write protection
- there is no access control for synchronisation

One way these problems can be overcome is by using *utility classes* to control locking.

Threads should periodically check their interrupt status, and if interrupted, shut down. A good place to check is before calling a synchronized method or calling `wait()`, e.g.,

```
// other code ...  
s.P();
```

as we may spend a long time contending for the lock on `s` and/or waiting in `P()` which can result in threads being unresponsive to interrupts.

```
if (Thread.interrupted()) throw new InterruptedException();
```

Synchronized methods and blocks limits use to strict block structured locking. We can't acquire a lock in one method or block and release it in another. However, we can implement more general mutual exclusion protocols using utility classes, e.g.:

```
class Mutex {  
    public void acquire() throws InterruptedException;  
    public void release();  
}
```

A `Mutex` can be used to replace blocks of the form:

`synchronized(lock) { body }` with the before/after construction:

```
mutex.acquire();  
/* body */  
mutex.release();
```

Mutex vs synchronized blocks

The two are not exactly equivalent. Unlike synchronized blocks, locking using the `Mutex` class is not reentrant - if the lock is already held by the thread performing the `acquire()`, it will deadlock. It is possible to define a `ReentrantLock` class, but the simple `Mutex` class is sufficient for many locking applications.

There is no way to alter the semantics of a Java synchronised lock, e.g., read vs write protection. This makes it difficult to solve selective mutual exclusion problems, like the Readers and Writers problem. Again, these problems can be overcome by using *utility classes* to control locking.

Read-Write locks are preferable to plain locks when the methods in a class can be separated into those that only read internally held data and those that write, when reading is not permitted while writing methods are executing, when the application has more readers than writers and when the methods are time consuming, so it pays to introduce more overhead in order to allow concurrency among reader threads.

Revision Notes

```
class ReadersWriters {
    protected int readers = 0;
    protected int waitingReaders = 0;
    protected boolean writing = false;
    protected int waitingWriters = 0;

    public synchronized void startRead() {
        waitingReaders++;
        while (writing || waitingWriters > 0) {
            try { wait(); }
            catch (InterruptedException ie) {
                waitingReaders--;
                Thread.currentThread().interrupt();
                return;
            }
        }
        waitingReaders--;
        readers++;
    }

    public synchronized void endRead() {
        readers--;
        notifyAll();
    }

    public synchronized void startwrite() {
        waitingWriters++;
        while (writing || readers > 0) {
            try { wait(); }
            catch (InterruptedException ie) {
                waitingWriters--;
                Thread.currentThread().interrupt();
                return;
            }
        }
        waitingWriters--;
        writing = true;
    }

    public synchronized void endwrite() {
        writing = false;
        notifyAll();
    }
}
```

All the synchronisation primitives we have looked at are equivalent in the sense that they all have the same expressive power. While it is often helpful to take advantage of the higher level of abstraction offered by monitors, there are situations when other forms of synchronisation are required and we can implement any of these using any of the primitives. More complex forms of locking can and are defined in terms of primitives like Mutex. At each level of abstraction we see this pattern of acquiring and releasing locks.

Condition Synchronisation can be implemented using the methods `wait()`, `notify()` and `notifyAll()`. To delay a thread until some condition is true, write a `wait()` loop that causes the thread to block if the delay condition is false. Ensure that every method which changes the truth value of the delay condition notifies threads waiting on the condition (using `notify()` or `notifyAll()`), causing them to wake up and re-check the delay condition.

When a thread blocks and/or another is scheduled, the JVM must perform a *context switch*. This involves saving the registers of the suspended thread and loading the registers of the newly scheduled thread which takes time. Therefore a concurrent program, runs faster if we can reduce the number of context switches.

In Java, each object has a *single* implicit condition variable. A `notifyAll()` intended to inform threads about one condition also wakes up threads waiting for unrelated conditions, resulting in large numbers of context switches. Context switching can be minimised by delegating operations with different `wait()` conditions to different *helper objects*. Such helper objects serve as *condition variables* - places to put threads that need to wait and be notified.

Bounded Buffer Implementations in Java

```
class BoundedBuffer {
    Object[] buf;
    int out = 0,          // index of first full slot
    int in = 0,          // index of first empty slot
    int count = 0;      // number of full slots

    public BoundedBuffer(int n) {
        buf = new Object[n];
    }

    public synchronized void append(Object data) {
        try { while(count == n) wait(); }
        catch (InterruptedException e) { return; }
        buf[in] = data;
        in = (in + 1) % n;
        count++;
        notifyAll();
    }

    public synchronized Object remove() {
        try { while(count == 0) wait(); }
        catch (InterruptedException e) { return null; }
        Object item = buf[out];
        out = (out + 1) % n;
        count--;
        notifyAll();
        return item;
    }
}

class BoundedBufferWithSemaphores {

    BufferArray buff;
    GeneralSemaphore empty;
    GeneralSemaphore full;

    public BoundedBufferWithSemaphores(int n) {
        buff = new BufferArray(n);
        empty = new GeneralSemaphore(n);
        full = new GeneralSemaphore(0);
    }

    public void append(Object data) throws InterruptedException {
        empty.P();
        buff.append(data);
        full.V();
    }

    public Object remove() throws InterruptedException {
        full.P();
        Object data = buff.remove();
        empty.V();
    }
}

class BufferArray {

    Object[] array;
    int in = 0, out = 0;

    BufferArray(int n) {
        array = new Object[n];
    }
}
```

Revision Notes

```

synchronized void append(Object data) {
    array[in] = data;
    in = (in + 1) % array.length;
}
synchronized Object remove() {
    Object data = array[out];
    array[out] = null;
    out = (out + 1) % array.length;
    return data;
}
}
}

```

The BoundedBufferWithSemaphores class is likely to run more efficiently than the BoundedBuffer class when many threads are using the buffer as BoundedBufferWithSemaphores uses two different underlying wait sets. The semaphores only wake one thread on each operation, eliminating the unnecessary context switching caused by using notifyAll() instead of notify() which reduces the worst case number of wakeups from a quadratic function of the number of invocations to linear.

Multiparty Actions

Although fully synchronised objects are always safe, threads using them are not always live. Some synchronized actions are multiparty – they acquire locks on multiple objects. *Deadlock* is possible when two or more objects are mutually accessible from two or more threads, and each thread holds one lock while trying to obtain another lock held by another thread.

```

class Cell {
    private long value;
    synchronized long getValue() { return value; }
    synchronized void setValue(long v) { value = v; }
    synchronized void swapValue(Cell other) {
        long t = getValue();
        long v = other.getValue();
        setValue(v);
        other.setValue(t);
    }
}

```

Example trace:

<u>Thread 1</u>	<u>Thread 2</u>
acquire lock for a on entering	acquire lock for b on entering
a.swapValue(b)	b.swapValue(a)
pass the lock for a (since already held)	pass lock for b (since already held) on
on entering t = getValue()	entering t = getValue()
block waiting for lock on b on entering	block waiting for lock on a on entering
v = other.getValue()	v = other.getValue()

Resource ordering

One way to avoid this kind of deadlock is to use resource ordering:

- associate a numerical (or any other strictly orderable data type) tag with each object that can be an argument to a synchronized multiparty action.
- if synchronization is always performed in tag order, then a situation can never arise in which a thread which has a lock on object x and is waiting for a lock on y while another thread has a lock on y and is waiting for a lock on x.
- whichever thread locks the resource with the lowest tag first will acquire both locks while the other waits, and then the second thread will acquire both locks

```

public void swapValue(Cell other) {
    if (System.identityHashCode(this) < System.identityHashCode(other))
        this.doSwapValue(other);
    else other.doSwapValue(this);
}

```

Message Passing

We can distinguish three types of implementations of concurrency. Multiprogramming (execution of concurrent processes by timesharing them on a single processor), Multiprocessing (the execution of concurrent processes by running them on separate processors which all access a shared memory) and distributed processing (the execution of concurrent processes by running them on separate processors which communicate by message passing).

Distributed processing

Network architectures, in which processors share only a communication network are becoming increasingly common: e.g., networks of workstations or multicomputers with distributed memory. The processes don't share a common address space, so they can't communicate via shared variables. Instead they communicate by *sending and receiving messages*.

Processes communicate by sending and receiving messages using special message passing primitives which include synchronisation:

- send <destination> <message>: sends message to another process destination
- receive <source> <message>: indicates that a process is ready to receive a message message from another process source.

No special mechanisms are required for mutual exclusion, but new techniques are needed for condition synchronisation.

Many different approaches to message passing have been proposed which differ in terms of whether communication is one way or two way, whether the naming of sources and destinations is direct or indirect, whether the naming of sources and destinations is symmetrical or asymmetrical and how send and receive operations are synchronised.

If communication is *one way* a process can only send or receive on a given channel in a single operation. We need to use two channels to establish two way communication between processes. All the message passing primitives we will look at below use *one way communication*.

The naming of the source and destination of messages can be either *direct* (using the names of processes) or *indirect* (using the names of channels) and either *symmetrical* (both **send** and **receive** operations name the processes or channels) or *asymmetrical* (only **send** names processes or channels and **receive** receives from any process or channel).

In *direct naming*, unique names are given to all processes comprising a program. In *symmetrical direct naming* both the sender and receiver name the corresponding process. In *asymmetrical direct naming* the receiver can receive messages from any process. *Indirect naming* uses intermediaries called channels or mailboxes. In *symmetrical indirect naming* both the sender and receiver name the corresponding channel. In *asymmetrical indirect naming* the receiver can receive messages from any channel. Below, we will use *asymmetrical indirect naming*.

Synchronising send and receive

If a process tries to receive a message before one has been sent by another process, it will block until there is a message for it to read. The differences are mainly in the behaviour of the sending process:

- asynchronous send : e.g., Unix sockets, Java.net
- synchronous send : e.g., CSP, occam
- remote invocation : e.g., extended rendezvous, RPC

If a process sends a message and continues executing without waiting for the message to be received, then the communication is termed *asynchronous*. Send operations are non-blocking so a sending process can get arbitrarily far ahead of a receiving process. Message delivery is not guaranteed if failures can occur and since channels can contain an unbounded number of messages, messages have to be buffered.

The receiving process cannot know anything about the current state of the sending process and the sending process has no way of knowing if the message was ever received unless the receiving process sends a reply. It is hard to detect when failures have occurred. Buffer space is finite - if too many messages are sent either the program will crash, the buffer will overflow with loss of messages, or send operation will block.

Revision Notes

Synchronous message passing

If the sending process is delayed until the corresponding receive is executed, the message passing is *synchronous*. Both the **send** and **receive** operations are blocking. A process sending to a channel delays until another process is ready to receive from that channel and so messages don't need to be buffered.

Synchronous message passing can result in reduced concurrency as whenever two processes communicate, at least one of them will have to block (whichever one tries to communicate first). Concurrency is also reduced in some client-server interactions, e.g., when a client is releasing a resource, there is usually no reason for it to wait until the server has received the release message. Similarly, when a client writes to a device (e.g., a file or graphics display) it can usually continue without waiting for the server to receive the message.

Example: Active monitors

```
// globally available names for n channels
channel request;
channel[] reply = new channel[n];

// Resource allocation process
process ResourceAllocator {

    list units = new list[MAXUNITS];
    queue pending;
    integer avail = MAXUNITS;
    integer clientID, unitID;

    while (true) {
        receive request <clientID, kind, unitID>;
        if (kind == ACQUIRE) {
            if (avail > 0) {
                avail--;
                remove(units, unitID);
                send reply[clientID] <unitID>
            } else insert(pending, clientID);
        } else { // kind == RELEASE
            if (empty(pending)) {
                avail++;
                insert(units, unitID);
            } else {
                remove(pending, clientID);
                send reply[clientID] <unitID>;
            }
        }
    }
}

// ith Client process of n ...
process Client {
    integer unitID;
    send request <i, ACQUIRE, null>;
    receive reply[i] <unitID>;
    // use the resource unitID, then release it ...
    send request <i, RELEASE, unitID>;
}
```

This is one way to program a simple monitor as an active process rather than a passive collection of procedures. We get mutual exclusion because only the server process can access its own local variables. The monitor procedures typically get turned into the branches of an if or switch statement, so only one 'monitor procedure' can be active at a time, and the monitor procedures run with mutual exclusion. Condition synchronisation is programmed with normal variables - conditions are re-evaluated on the arrival of a new message.

Example: Scheduling Disk Access

Data is stored in *blocks* which are arranged in concentric *tracks*. Tracks in the same relative position on different platters form a *cylinder*. The address of a block consists of a cylinder, track and offset from a start point. Data is accessed by positioning a read/write head over the appropriate track and waiting until the required block passes under the head.

Disk access time depends on:

- the seek time to move the heads to the appropriate cylinder
- the time required to move the heads depends on the distance between the two cylinders - seeking even one cylinder takes much longer than the maximum rotational delay. To reduce the *average* disk access time, we need to minimise head motion and hence seek time.
- the rotational delay waiting for the block to pass under the heads
- the data transmission time

Scheduling disk requests

The key idea is to service the disk requests *out of order* when two or more clients want to access the disk at about the same time. Suppose we have a sequence of requests for cylinders 100, 1, 101, 2, ... with the heads initially at cylinder 100. If we process these in the order of arrival, the heads travel a total of 300 cylinders. If we process them in the order 100, 101, 2, 1..., the heads travel a total of 101 cylinders. Furthermore the client requesting cylinder 1 only has to wait for the heads to move an extra two cylinders.

Message passing Solution

```
// shared channels
channel request;
channel[] reply = new channel[n];

// ith Client process of n ...
process Clienti {
    disk_block dblock;
    integer track, offset;

    // request a block of data from the disk
    send request <i, track, offset>;
    receive reply[i] <dblock>;
    // use the data from the disk ...
}

// Self-Scheduling Disk Driver process
process DiskDriver {

    queue saved;
    disk_block dblock;
    integer clientID, track, offset, nsaved = 0;
    while(true) {

        while(!empty(request) or nsaved == 0) {

            // wait for first request or receive another
            receive request <clientID, track, offset>;
            insert(saved, <clientID, track, offset>);
            nsaved++;

        }

        // select the pending request closest to the current head position and
        access the disk
        remove(saved, <clientID, track, offset>);
        nsaved--;
        dblock = read(track, offset);
        send reply[clientID] <dblock>;

    }
}
```

Revision Notes

On a shared memory machine, procedure calls and operations on condition variables are more efficient than message passing primitives. Most distributed systems are based on message passing since it is more natural and more efficient than simulating shared memory on a distributed memory machine. Neither asynchronous nor synchronous message passing have yet found their way into a widely accepted general purpose programming language. Message passing in concurrent programs remains at the level of OS or library calls.

Remote Invocation

Both asynchronous and synchronous message passing assume *one-way* communication. Messages are transmitted in one direction only, from sender to receiver. Message passing is well suited to problems in which the flow of information is essentially one-way, e.g., producer-consumer problems. Two way information flow between, e.g., between clients and servers, has to be programmed with two explicit message exchanges using two different channels.

With *remote invocation* a process executes a synchronous send and waits until the reply is received. This combines aspects of *monitors* and *synchronous message passing*. As with monitors, interaction is via public procedures and as with synchronous send, calling a procedure delays the caller. This provides a *two way* communication channel from the caller to the process servicing the call and back. It is implemented using message passing.

There are two main forms of remote invocation:

- *Remote Procedure Call* creates a *new* process to handle each call
- *Extended Rendezvous* services a request using an *existing* process

Modules

A module contains both processes and local and exported procedures.

The *header* contains the signatures of the exported procedures, the *body* contains local procedures and processes, local variables, and initialisation code. At any point in time, a module contains zero or more *processes*. Different modules may reside in different addresses spaces.

Modules and message passing

Communication *between* modules is by calls to exported procedures. Arguments and return values are passed as *messages*. The sending and receiving of messages is *implicit* rather than explicitly programmed. Communication *within* modules is similar to monitors: processes within a module can share variables and call procedures declared in that module.

Modules and RPC

In RPC, a module contains *zero or more* processes and some exported procedures. Local processes are called *background processes*. Processes that result from remote calls to exported procedures which are called *server processes*.

Synchronisation in modules

There are two ways for server and background processes in an RPC module to have mutually exclusive access to shared variables and to synchronise with each other.

- all the processes in the same module execute with mutual exclusion (as in monitors) – condition synchronisation is programmed explicitly using *semaphores* and/or *condition variables*.
- processes execute concurrently within a module and both mutual exclusion and condition synchronisation are programmed explicitly using *semaphores* and/or *condition variables*.

Example: Time Server

A *time server* provides timing services to client processes.

```
module TimeServer
  export integer get_time();
  export void delay(integer interval);
body
  integer time = 0;
  binary semaphore m = 1;
  binary semaphore[] d = new binary semaphore[n] {0};
  queue napQ;

  integer get_time() {
    return time;
  }

  void delay(integer interval) {
    integer waketime = time + interval;
    P(m);
    insert(napQ, <waketime, serverProcID>);
    V(m);
    P(d[serverProcID]);
  }

  // background Clock process ...
  process Clock {
    // start hardware timer (omitted) ...
    while(true) {
      // wait for interrupt then restart timer...
      time++;
      P(m);
      while(time >= smallest waketime on napQ) {
        remove(napQ, <waketime, serverProcID>);
        V(d[serverProcID]);
      }
      V(m);
    }
  }
}
```

Extended rendezvous

Extended rendezvous combines *communication* and *synchronisation*:

- as with RPC a process invokes an operation by means of a remote call:
 - o a server process waits for and then acts on a single call
 - o calls are serviced one at a time rather than concurrently
- the caller and server processes synchronise (rendezvous) on the call

Modules and extended rendezvous

In extended rendezvous a module contains a *single* process and some exported operations. The header contains signatures of *operations* (or entry points) exported by the module. The body of a module consists of a *single* process that services the call. *Accept* statements block the server process until there is at least one pending call of an exported operation. As with RPC, arguments to the call and any return values are passed as messages between the caller and server processes.

Example: Ada Rendezvous

In Ada, a module is called a *task* and exported operations are called *entries*. The body of a task contains variable and procedure declarations and the program statements executed by the task. For each entry declared in the header, there is a corresponding accept statement. Execution of a task blocks at an accept statement, unless there is a call on the corresponding entry. When there is a call on the entry, the statements that make up the body of the accept statements are executed, and any results returned.

Revision Notes

Ada entry and accept statements:

```
task <name> is
  entry <entryID1>(args);
  entry <entryID2>(args);
end;

task body <name> is
  // local declarations
  begin
  // statements
  loop
    select
      accept <entryID1>(args) do
        // statements
      end;
    or
      accept <entryID2>(args) do
        // statements
      end;
    end select;
  end loop;
  // more statements
end <name>;
```

For example, a call to an entry called ACQUIRE in a task called ResourceAllocator has the form
call ResourceAllocator.ACQUIRE(args);

and is serviced by an accept statement in the body of ResourceAllocator of the form:

```
accept ACQUIRE(args) do
  // statements to process the ACQUIRE request
end;
```

```
while (true) {
  receive request <clientID, kind, unitID>;
  if (kind == ACQUIRE) {
    if (avail > 0) {
      avail--;
      remove(units, unitID);
      send reply[clientID] <unitID>
    } else insert(pending, clientID);
  } else // kind == RELEASE
    // free a unit of resource ...
}
```

```
task ResourceAllocator is
  entry ACQUIRE(args);
  entry RELEASE(args);
end;
```

```
task body ResourceAllocator is
  // declaration list of free units, pending queue etc.
  begin
  loop
    select
      accept ACQUIRE(args) do
        // process the ACQUIRE request
      end;
    or
      accept RELEASE(args) do
        // process the RELEASE request
      end;
    end select;
  end loop;
end ResourceAllocator;
```

Revision Notes

Distributed Processing in Java

The package `java.net` supports *asynchronous* message passing: Naming is indirect and symmetrical. Communication is one way - a process can only send or receive on a given channel in a single operation. Receiving is synchronous, sending is asynchronous.

Ports

Application processes are identified by their transport protocol (TCP or UDP) and *port number(s)* (16 bit values). E.g., all systems that offer Telnet do so on port 23. The source port number identifies the process that sent the data and destination port number identifies the process that will receive the data. Dynamically allocated ports are assigned when needed, e.g., for Telnet connections the source port is assigned a dynamically allocated port number and port 23 is used for the destination port. It is the *pair* of port numbers that uniquely identifies each network connection.

Sockets

The combination of an IP address and a port number is called a *socket*. A socket is one end of a two-way communication link between two or more processes. Each pair of sockets provides 2 unidirectional channels (streams) where communication is one-way - a process can only send or receive on a given channel (stream) in a single operation. Cf remote invocation, where a process both sends and receives on the same channel in a single method invocation.

Transport layer protocols

`java.net` provides support for two transport layer protocols. TCP provides reliable, ordered, buffered (one-way) communication between two processes, e.g., HTTP, FTP, Telnet etc. (Socket, ServerSocket). UDP sends independent packets of data (datagrams) from one process to another with no guarantees about reliability or ordering, e.g., ping, time services etc. (DatagramSocket, MulticastSocket).

`java.net` TCP

The Socket and ServerSocket classes provide reliable, ordered, buffered communication between two processes. Communication is point-to-point - each Socket and ServerSocket provide two unidirectional byte streams, an InputStream and an OutputStream. Naming is indirect and symmetrical - clients name the host and port to which they wish to connect, servers accept connections from any client on that port and then create a new socket for the client. Reads are synchronous: reading from an InputStream causes the reading process to block and writing is asynchronous: the message is buffered by the OutputStream.

`java.net` UDP

The DatagramSocket and MulticastSocket classes provide 'best effort' communication between two or more processes. Communication is point-to-point (DatagramSocket to DatagramSocket) or broadcast (DatagramSocket to MulticastSocket) and message based. Naming is indirect and symmetrical - clients name the host and port to which they wish to send a message as part of the datagram, servers accept messages from any client on that port and can also send messages to a group of clients listening on a particular port. Reads are synchronous: receiving from the socket causes the receiving process to block and writing is asynchronous.

ServerSocket

The `accept()` method waits until a client requests a connection to the host and port of this server. When the connection is successfully established, the `accept()` method returns a *new* Socket object which is bound to a *new* (dynamically allocated) port. The server can communicate with the client over this new port while continuing to listen for client connection requests on the ServerSocket on the original port. This implies the server is multi-threaded.

Supporting multiple clients

The KnockKnockServer in the *Java Tutorial* processes a single connection request (client) and then exits. In order to allow the server to handle multiple clients, we can create a new thread for each client connection:

```
while (true) {  
    accept a connection ;  
    create a thread to deal with the client ;  
}
```

The thread reads from and writes to the client connection as necessary

See Java Tutorials et al. for code samples.

java.rmi

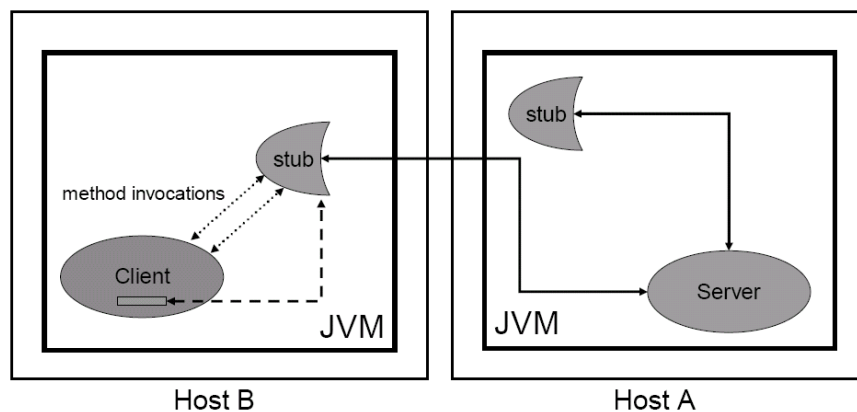
The package `java.rmi` implements Java's version of RPC. Remote invocation is based on the model of a *procedure call*. In Java, non-static methods must be invoked on an *object*, Java therefore requires both *remote methods* (procedures) and *remote objects* on which the remote methods can be invoked.

A Java *remote object* is one whose methods can be invoked from another JVM, potentially on a different host. A remote object is described by one or more *remote interfaces* which extend `java.rmi.Remote`. *Remote method invocation* (RMI) is the action of invoking a method of a remote interface on a remote object.

A Remote interface is similar to the *header* of a module containing the signatures of the exported procedures. A class implementing a Remote interface is similar to the *body* of the module, containing local methods and variables, and initialisation code. The *processes* in a module are the threads running on the target JVM. Details of communication between Remote objects are handled by RMI, which uses Sockets and Serialization to implement the transfer of arguments and results.

An RMI application

A *server* application creates some remote objects, makes references to them accessible and waits for clients to invoke (remote) methods on the remote objects. A *client* gets a remote reference to a remote object in the server, either from the RMI registry or as a return value to a remote method, and invokes (remote) methods on it. A component of a distributed Java application can act as both a client and server.



Stubs

A *stub* acts as a proxy for a remote object and is responsible for carrying out method calls on the remote object. Invoking a stub method:

- initiates a connection with the remote JVM containing the remote object
- writes and transmits the method parameters to the remote JVM
- waits for the results of the method invocation and
- reads the result (return value or exception) and returns it to the caller.

Skeletons (Java 1 only)

In the remote JVM each remote object may have a corresponding *skeleton*. When a skeleton receives an incoming method invocation, it:

- reads the parameters for the remote method
- invokes the method on the actual remote method implementation and
- writes and transmits the result (return value or exception) to the stub

Stubs and skeletons are generated by the Java RMI stub compiler, `rmic`. For communication between Java 2 platforms, skeletons are not used.

Revision Notes

Parameter passing in RMI

An argument to or return value from a remote object can be any Java object that is *serializable*. Non-remote method arguments and results are passed by *copying* - changes made to the object are not visible to other clients. Remote objects are passed by *reference* (i.e., a copy of the stub is passed or returned) - changes made by one client to the state of the remote object are visible to all clients.

RMIC

Stubs (and skeletons if required) are generated by the Java RMI Stub compiler, *rmic*. The *rmic* command takes one or more class names as input and produces as output class files of the form `ClassName_Stub.class` and `ClassName_Skel.class` (a skeleton file will not be generated if you run *rmic* with the `-v1.2` option, e.g., if all of your clients will be running JDK 1.2 or compatible versions).

Example: Savings Account in Java

```
class SavingsAccount {
    protected long balance;
    public SavingsAccount (long openingBalance) { balance = openingBalance; }
    public synchronized long balance() { return balance; }
    public synchronized void deposit(int amount) {
        balance += amount;
        notifyAll();
    }
    public synchronized void withdraw(int amount) throws InterruptedException {
        while (amount > balance) wait();
        balance -= amount;
    }
}
```

Example: Remote Savings Account

```
import java.rmi.*;
import java.rmi.server.*;

public interface RemoteSavingsAccount extends Remote {
    public long balance() throws RemoteException;
    public void deposit(int amount) throws RemoteException;
    public void withdraw(int amount) throws RemoteException, InterruptedException;
}

class RemoteSavingsAccountServer extends UnicastRemoteObject implements
    RemoteSavingsAccount {
    protected long balance = 0;
    public RemoteSavingsAccountServer(int openingBalance) throws RemoteException {
        balance = openingBalance;
    }

    public synchronized long balance() throws RemoteException { return balance; }

    public synchronized void deposit(int amount) throws RemoteException {
        balance += amount;
        notifyAll();
    }

    public synchronized void withdraw(int amount) throws RemoteException,
        InterruptedException {
        while (amount > balance) wait();
        balance -= amount;
    }

    public static void main(String[] args) {
        try {
            RemoteSavingsAccount server = new RemoteSavingsAccountServer(100);
            Naming.bind("//host:port/savings", server);
        } catch (Exception e) { System.err.println(e); }
    }
}
```

Revision Notes

The main method creates an instance of the `RemoteSavingsAccountServer`. This calls the `UnicastRemoteObject` constructor which in turn exports the newly created object to the RMI runtime. Before a caller can invoke a method on a remote object, it must obtain a remote reference to it - the `Naming` interface is used for registering and looking up remote objects in the *registry*. Once this has happened, the `RemoteSavingsAccount` remote object is ready to accept incoming calls from clients on an anonymous port chosen by RMI or the underlying OS. The main method then exits - as long as there is a reference to the `RemoteSavingsAccount` object in another JVM, on the same or a different host, the JVM will not be shut down.

Clients

```
// Deposits a specified number of random amounts)
class DepositClient {

    public static void main(String[] args) {
        try {
            System.setSecurityManager(new RMISecurityManager());
            String name = "//host:port/savings";
            RemoteSavingsAccount s =(RemoteSavingsAccount) Naming.lookup(name);
            Random rng = new Random();
            int deposits = Integer.parseInt(args[0]);
            for (int i = 0; i < deposits; i++) s.deposit(rng.nextInt(10) + 1);
        }
        catch (Exception e) {
            System.err.println(e);
        }
    }
}

class WithdrawClient {

    public static void main(String[] args) {
        try {
            System.setSecurityManager(new RMISecurityManager());
            String name = "//host:port/savings";
            RemoteSavingsAccount s =(RemoteSavingsAccount) Naming.lookup(name);
            Random rng = new Random();
            int withdrawals = Integer.parseInt(args[0]);
            for (int i = 0; i < withdrawals; i++)
                s.withdraw(rng.nextInt(50) + 1);
        }
        catch (Exception e) {
            System.err.println(e);
        }
    }
}
```

The RMI registry

The system provides a particular remote object, the *RMI registry* for finding references to remote objects. Once a remote object is registered with the RMI registry on the local host, clients on any host can look up the remote object by name, obtain a reference to it (stub), and then invoke its methods. The registry listens on a known port, usually 1099 on the same host as the server.

RMI and Threads

"A method dispatched by the RMI runtime to a remote object may or may not execute in a separate thread. The RMI runtime makes no guarantees with respect to mapping remote method invocations to threads. Since remote method invocation on the same remote method may execute concurrently, a remote object implementation needs to make sure its implementation is thread-safe."

- Java RMI Specification, section 3.2

Proving Correctness

Remember that the protocols should satisfy the following properties:

- Mutual Exclusion: at most one process at a time is executing its critical section.
- Absence of Deadlock (Livelock): if two or more processes are attempting to enter their critical sections, at least one will succeed.
- Absence of Unnecessary Delay: if a process is trying to enter its critical section and other processes are executing their noncritical sections (or have terminated), the first process is not prevented from entering its critical section.
- Eventual Entry: a process that is attempting to enter its critical section will eventually succeed.

Finding bugs

How can we determine if an algorithm satisfies these properties? If an algorithm is broken, it is often relatively easy to find a trace which violates one or more of the properties, however showing that there is no such trace is much harder. Non-exhaustive testing can only show the existence of bugs, not their absence.

Demonstrating correctness

Testing can only consider a limited number of program executions. Some logically possible interleavings may not be generated by a particular implementation and so the only way to ensure that a concurrent program is correct is to prove that it is by proving that certain properties are true of all executions of the program.

Proving Correctness

- Assertional reasoning: involves using assertions and invariants specified in predicate logic.
- Model checking: involves showing that a program represented as a finite state machine or a labelled transition system is a valid model of a formula expressing the desired property.

Proving mutual exclusion of Dekker's algorithm

We need to show that Process 1 in crit1 implies Process 2 is not in crit2

1. When Process 1 entered crit1, c2 was not 0.
2. c2 is not 0 implies that Process 2 is not in crit2.
3. When Process 1 entered crit1 then Process 2 was not in crit2.
4. Process 1 in crit1 implies c1 is 0.
5. c1 is 0 implies Process 2 does not enter crit2.
6. Process 1 in crit1 implies Process 2 does not enter crit2.
7. As long as Process 1 in crit1, Process 2 will never enter crit2.
8. Process 1 in crit1 implies Process 2 is not in crit2.

By symmetry, Process 2 in crit2 implies that Process 1 is not in crit1

The proof is in two parts:

- we first show that Process 1 can't enter its critical section if Process 2 is in its critical section (steps 1–3); and
- having shown the Process 2 can't be in its critical section when Process 1 enters its critical section, we show that this continues to hold for all future time points in which Process 1 remains in its critical section - Process 2 can't enter its critical section while Process 1 is in its critical section (steps 4–8).

By symmetry, Process 2 in crit2 implies that Process 1 is not in crit1 and we have established Mutual Exclusion.

Revision Notes

Proving eventual entry

```

// Process 1
init1;
while(true) {
 $\alpha_2$  c1 = 0; // entry protocol
 $\alpha_3$  while (c2 == 0) {
 $\alpha_4$    if (turn == 2) {
 $\alpha_6$        c1 = 1;
 $\alpha_8$        while (turn == 2) {};
           c1 = 0;
       }
}
 $\alpha_5$  crit1;
 $\alpha_7$  turn = 2; // exit protocol
    c1 = 1;
 $\alpha_1$  rem1;
}

// Process 2
init2;
while(true) {
 $\beta_2$  c2 = 0; // entry protocol
 $\beta_3$  while (c1 == 0) {
 $\beta_4$    if (turn == 1) {
 $\beta_6$        c2 = 1;
 $\beta_8$        while (turn == 1) {};
           c2 = 0;
       }
}
 $\beta_5$  crit2;
 $\beta_7$  turn = 1; // exit protocol
    c2 = 1;
 $\beta_1$  rem2;
}

```

Theorem: α_2 and never α_5 is false, i.e. α_2 implies eventually α_5 .

Assumption: to simplify the proof, we assume that the processes do not terminate, either in their critical region or in the remainder. We also make use of the following invariants:

I₁ c1 == 0 if and only if α_3 or α_4 or α_5 or α_6 or α_7 .

I₂ c2 == 0 if and only if β_3 or β_4 or β_5 or β_6 or β_7 .

Since the only assignments to c1 are in Process 1 and to c2 are in Process 2, we can deduce the values of c1 and c2 from the α s and the β s respectively.

The proof is in two parts. We first show that (α_2 and never α_5) leads to a contradiction when (turn held at 2) and then show that (α_2 and never α_5) leads to a contradiction when (turn == 1). Since (α_2 and never α_5) leads to a contradiction in all cases, it must be false.

1. α_2 and never α_5 (Assumption)
2. turn held at 2 (Assumption)
3. eventually c1 held at 1 (From (1) and (2))
Since (never α_5), Process 1 eventually passes α_3 and α_4 . Since (turn held at 2), Process 1 reaches α_6 and α_8 and is then blocked in the loop at α_8 . By I₁, as long as Process 1 is at α_8 , c1 must equal 1.
4. eventually turn == 1 (From (2) and (3))
If (turn held at 2) and (c1 held at 1) and the processes do not terminate, then Process 2 must eventually reach β_7 assign the value 1 to turn.
5. Contradiction (From (2) and (4))
If (turn held at 2) and eventually (turn == 1) means that there is a time when turn is simultaneously 1 and 2. From (α_2 and never α_5) and (turn held at 2) we have deduced a contradiction.
6. eventually turn == 1
It follows that (α_2 and never α_5) implies eventually that (turn is not 2). Since turn must equal 1 or 2, we have proved (α_2 and never α_5) implies eventually (turn == 1). We have shown that if Process 1 reaches its entry protocol, but never succeeds in entering its critical section, then turn must eventually take the value 1.
7. never α_5 and never α_7 and never α_1 (From (1))
The only way to reach α_1 or α_7 from α_2 is to pass through α_5 but we assume that we never reach α_5 .
8. eventually turn held at 1 (From (6) and (7))
Once the value of turn is 1 (as ensured by step (6)) the only way that the value can change back to 2 is to execute α_7 . By step (7), this will never happen.
9. eventually Process 1 loops forever at $\alpha_3 - \alpha_4$ and c2 held at 0 (From (7) and (8))
By steps (7) and (8) we know that (Process 1 is never at α_5) and (turn held at 1). Therefore since Process 1 must reach $\alpha_3 - \alpha_4$ from α_2 , α_6 or α_8 it will then loop forever at $\alpha_3 - \alpha_4$.
10. eventually c1 held at 0 (From (9))
From step (9), eventually we loop at $\alpha_3 - \alpha_4$, which by I₁ implies that c1 will be held at 0.
11. eventually c2 held at 1 (From (8) and (10))
By a similar argument to step (10), Process 2 must eventually loop at β_8 . Then by I₂, c2 is held at 1.
12. Contradiction (From (9) and (11))
From steps (8) and (10) we have shown that (α_2 and never α_5) implies eventually (c2 held at 1), but this contradicts (9) - if c2 is held at 1, the Process 1 cannot be looping at $\alpha_3 - \alpha_4$. From (α_2 and never α_5) we have deduced a contradiction. Therefore (α_2 and never α_5) is false.
13. α_2 implies eventually α_5

Revision Notes

Model Checking

Formal verification consists of three parts:

- a description language for describing the system to be verified
- a specification language for describing the properties to be verified and
- a verification method to establish whether the description of the system satisfies the specification

In a *proof-based approach*,

- the system description is a set of formulas Γ in some logic
- the specification is another formula φ in the same logic
- the verification method consists of trying to find a proof that $\Gamma \vdash \varphi$

This is time consuming and requires expertise on the part of the user.

In a *model-based approach*,

- the system is represented by a finite model M for an appropriate logic
- the specification is a formula φ in the same logic and
- the verification method consists of computing whether M satisfies φ ($M \models \varphi$)

This process can be automated (model checking).

Model checking is an automatic, model-based, property verification approach, i.e., the specification describes a single property of the system rather than its complete behaviour. It is intended for concurrent, reactive systems, e.g., operating systems, embedded systems and computer hardware.

To verify that a program or system satisfies a property, we

- describe the system using the description language of the model checker
- express the property to be verified using the specification language of the model checker and
- run the model checker with the system description and property to be verified as inputs.

Model checking is based on temporal logic. In classical (propositional) logic, a model is an assignment of truth values to atomic propositions. The models of temporal logic contain several states and a formula can be true in some states and false in others. Truth is dynamic in that formulas can change their truth values as the system evolves from state to state. In model checking, the models M are transition systems and the properties φ are formulas of temporal logic.

When the model checker is run, it generates a model (transition system), M , from the system description, it then converts the property to be verified into a temporal logic formula φ and for every state s in M , checks whether s satisfies φ ($M, s \models \varphi$). If the model doesn't satisfy the formula most model checkers also output a trace of the system behaviour that causes the failure.

A *transition system* consists of a set of states and the transitions between them (a directed graph). The states are the states of the system being modelled. States are labelled by a set of atomic propositions which are true in that state, e.g., "variable x has value 1", "process 1 is in its critical section", etc. The transitions correspond to the atomic transitions of the system, e.g., atomic instructions or synchronized methods. There may be many transitions from each state - one for each process that could go next in an interleaving.

Example: simple transition system

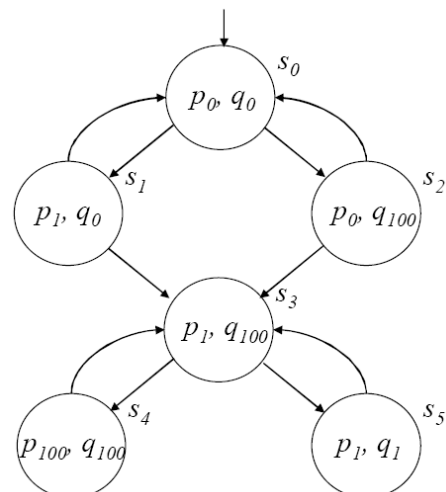
```
// shared variables
integer x = 0; y = 0;
```

```
// Process 1
while(true) {
x = 1;
x = y;
}
```

```
// Process 2
while(true) {
y = 100;
y = x;
}
```

Atomic propositions:
 p_0 true when $x == 0$
 p_1 true when $x == 1$
 p_{100} true when $x == 100$

q_0 true when $y == 0$
 q_1 true when $y == 1$
 q_{100} true when $y == 100$



Revision Notes

Model checkers don't usually take program text as input. A system description at the program statement level may be too fine grained for the properties to be checked. Model checkers are also used to verify hardware systems, communication protocols, etc. Instead, each model checker has its own description language and specification language.

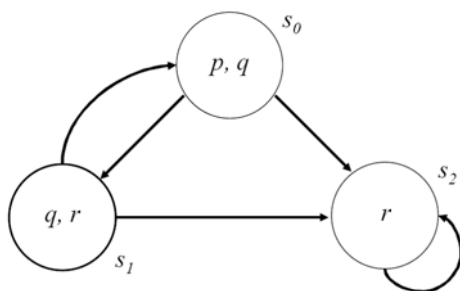
The property of the system to be verified is expressed in the model checker's specification language. Many model checkers allow properties to be expressed directly in temporal logic (often using a simplified syntax), for example, the SMV model checker uses Computation Tree Logic (CTL) as its specification language:

- a set of atomic propositions p, q, r, \dots
- standard logical connectives: $\neg, \square, \square, \rightarrow$
- temporal connectives: AX, EX, AF, EF, AG, EG, AU and EU
- formulas: $\varphi = p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \dots AX \varphi \dots A[\varphi U \phi] \dots$

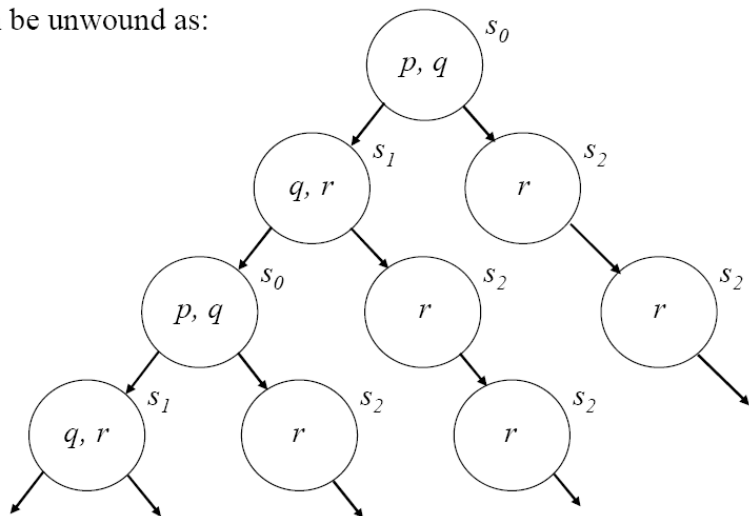
Temporal connectives:

- AX φ : on All paths, φ is true in the neXt state
- EX φ : on someE path, φ is true in the neXt state
- AF φ : on All paths, in some Future state φ is true
- EF φ : on someE path, in some Future state φ is true
- AG φ : on All paths, in all future states (Globally) φ is true
- EG φ : on someE path, in all future states (Globally) φ is true
- A $[\varphi U \phi]$: on All paths, φ is true Until ϕ is true
- E $[\varphi U \phi]$: on someE path, φ is true Until ϕ is true

CTL formulas can be evaluated relative to the computation tree which is the unwinding of the transition system describing the system. For example:



Can be unwound as:



- $M, s \models AX \varphi$: in every next state starting in s φ holds
- $M, s \models EX \varphi$: in some next state starting in s φ holds
- $M, s \models AF \varphi$: for all computation paths starting in s there is some future state where φ holds
- $M, s \models EF \varphi$: there exists a computation path starting in s such that φ holds in some future state
- $M, s \models AG \varphi$: for all computation paths starting in s the property φ holds globally (in every state along the path including s)
- $M, s \models EG \varphi$: there exists a computation path starting in s such that φ holds globally (in every state along the path including s)
- $M, s \models A[\varphi_1 U \varphi_2]$: for all computation paths starting in s the property φ_1 holds in every state along the path (including s) until φ_2 holds
- $M, s \models E[\varphi_1 U \varphi_2]$: there exists a computation path starting in s such that the property φ_1 holds in every state along the path (including s) until φ_2 holds