1. Multi-exit loop: Has one or more exit locations within a loop.

2.Static multi-level exit: exits multiple control structures where exit points are known at arbitrary location in a program.

3.Flag variable: used solely to affect control flow. (equivalent to a go to)

4. Situations where **dynamic allocation** (heap) is necessary: 1) outlive the block in which it is allocated 2) when the amount of data is unknown 3) array of objects must be initialized via constructor 4) large local variables are allocated on a small stack

5.Dynamic multi-level exit: extend call/return semantics to transfer in reverse direction to normal calls.

6. goto (non-local transfer) 1) find the stack 2) go to the label within the routine

7. fix-up routine: a routine called for an exception to fix-up and return to continue.

8. A return union of traditional approaches contains 1) result 2) return code

9. Drawbacks of traditional techniques:

   a. Error checking optional.
   b. Mix of error code and normal value
   c. Difficult to read and can be inappropriate
   d. Increase number of parameter.
   e. Status flag can be overwritten before examined.

10.Exception event: support of an algorithm which is known to exist, occurs with low frequency.

11. Exception: generated by executing an operation indicating an ancillary situation in execution.

12. Propagation: directs control from a raise in the source execution to a handler in the faulting execution.

13. Stack unwinding: all blocks on the faulting stack from the raise block to the guarded block handling the exception are terminated.

return/handled	call/raise	
	static	dynamic
static	sequel	termination exception
dynamic	routine	routine pointer, virtual routine, resumption

14. Resumption: control transfer from the start of propagation to handler, it do some fix-up and returns to the raise point (dynamic return), stack is not unwound.

15. No break, continue, return in **_CatchResume** handler. Why?

16. Coroutine: A routine can be suspended and resumed at soAme point, has own stack.

Usage: Eliminate computation or flag variables retaining information about execution state

17. Semi-coroutine：acts asymmetrically, non-recursive routine.

18. Full-coroutine: acts symmetrically, like recursive routines. (activation cycle)

Three phases:

a. Starting the cycle
b. Executing the cycle
c. Stop the cycle

19. Thread: an independent sequential execution path through program.

20. Process: a program component that has its own thread. (like routine)

21. Task: similar to a process but process has its own memory while tasks share a common memory. (also called a light-weight-process LWP)

22. Parallel execution: when 2 or more operations occurs simultaneously. (only in multi-core pc)

23. Concurrent execution: situation that multiple thread appears to be performed in parallel.

24. Advantage of concurrent:

1) Dividing a problem into multiple executing threads
2) Expressing a problem with multiple executing threads may be the natural way.
3) Time efficiency.

25. Speedup: Sc = T1/Tc

26. Amdahl's law: Sc = 1 / ((1 - P) + P/ C) where T1 = 1 and Tc = sequential + concurrent

27 **independent execution**: all threads created together and do not interact

dependent execution: threads created at different times and interact.

(longest path bounds speedup)

28. Critical section: A group of instructions on an associated object that must be performed atomically.

29. Mutual exclusion: preventing simultaneous execution of a critical section by multiple threads.

30. Rule of the mutual exclusion

1) only one thread in a critical section at a time (safety)

2) thread runs arbitrary speed and order
3) if thread not in (or entering) critical section, it must not prevent other thread entering critical section
4) do not cause livelock
5) no starvation, must eventually enter.

31. Unfairness: threads are not serviced in first-come first-serve order.

32. Barging: a waiting thread is overtaken by a thread arriving later.

33 Dekker algorithm: declare intent and check if other thread want in. If other does not want in, just go in. Otherwise, who goes in last time have lower priority, and retreat intent.

34. Peterson algorithm: two threads race to write in to an address at the same time, whoes value is overwritten is the one goes in. (RW-unsafe)

35. RW-safe: mutual exclusion works if simultaneous writes scramble bits and simultaneous read/write reads flickering bits.

36. SpinLock: busy waiting

37. Blocking Lock: reduce busy waiting by cooperation (releasing lock wakes blocking thread)

38. Single acquisition VS Multiple acquisition

39.Lock release pattern:

a. Executable statement (Final)
b. RAII

40. Synchronization Lock: used to block tasks waiting for synchronization (conditional variable)

external locking: use an external lock to protect task list

internal locking: use an internal lock to protect state

41. uCondLock (synchronization lock for uC++)

wait atomically blocks the calling task and releases the argument owner-lock

42. Barrier: coordinates a group of tasks performing a concurrent operation surrounded by sequential operations (for synchronization cannot build mutual exclusion)

43. Binary Semaphore: 0 = closed, 1 = open; P = acquire = -1; V = release = + 1

44. Counting Semaphore: allow multi-valued semaphore

45. rules of **baton passing** are**:**

- there is exactly one (conceptual) baton
- nobody moves in the entry/exit code unless they have it
- once the baton is released, cannot read/write variables in entry/exit

46. Why mutex/condition lock cannot perform baton passing

Signaller must release the mutex lock.

47. Read and Writer problem

a. Solution 1: Writer Starvation.
b. Solution 2: Reader Starvation.
c. Solution 3: Freshness
d. Solution 4: use shadow queue to avoid freshness and staleness.
e. Solution 5: use chair for writer instead of shadow queue.
f. Solution 6: interrupt
    i. Atomic block
    ii. Ticket
    iii. Private semaphore
g. Staleness: Reader reads data that is stale (old)
h. Freshness: Reader reads data that is too fresh(writer overwrites old data).

48. A **race condition** occurs when there is missin: (can be very difficult to locate)

- synchronization
- mutual exclusion

49. No progress:
    a. **Live lock** (always exists some mechanism to break tie)
    b. **starvation** (some tasks are ignored and never executed):
    c. **Deadlock**: state when one or more processes are waiting for an event that will not occur
        i. Synchronization Deadlock (failure in cooperation, so a blocked task is never unblocked)
        ii. Mutual Exclusion Deadlock (failure to acquire a resource protected by mutual exclusion)
50. 5 conditions that must occur for a set of processes to get into **Deadlock**
    1. exists > 1 shared resource
    2. hold and wait
    3. no preemption
    4. exists a circular wait of processes on resources
    5. these conditions must occur simultaneously
51. Deadlock prevention
    a. Synchronization Prevention
        i. No communication (completely independent)
    b. Mutual Exclusion Prevention
        i. no mutual exclusion (impossible in many cases)
        ii. no hold and wait (poor resource utilization and possible starvation)

iii. allow preemption (preemption is dynamic => cannot apply statically)

iv. no circular wait (use **ordered resource,** in some cases, some process are forced to acquire resources in an unnatural sequence or unwanted resource)

1. prevent simultaneous occurrence (show previous 4 rules cannot happen simultaneously.)

52. Deadlock Avoidance

a. Banker's Algorithm

i. Every process must state its maximum resource needs

ii. Calculate remaining resource .

iii. Give all resource to the one can run. After it finishes, all resource are returned

b. Allocation graph

i. No cycle -> No deadlock

ii. Create isomorphic graph without multiple instance -> expensive difficult

iii. Release one task may make the cycle broken.

53. **Detection and Recovery**

- Instead of avoiding, let it happen and recover (ability to discover deadlock and preemption)
- Discovering deadlock is not easy (eg. build and check for cycles)
- Recovery involves preemption of one or more processes in a cycle. (must prevent starvation, preemption victim must be restarted)

54. **Critical region**

- Mutual exclusion guaranteed
- Access within the REGION

55. **Monitor** : Abstract data type that combines shared data with serialization of its modification (recursive entry is allowed)

- **Mutex member**: does **not** begin execution if there is another active mutex member

56. **External Scheduling**: _Accept controls which mutex members can be accept calls, an acceptor blocks until a call to specified mutex member(s) occurs. When the accepted task exits, the acceptor continues.

57. **Internal Scheduling**: Scheduling among tasks inside the monitor. A condition(uCondition) is a queue of waiting tasks.

- wait(): blocks the current thread and restarts a signalled task
- signal(): unblocks the thread on the front of the condition queue
- signalBlock(): unblocks the thread on the front of the condition queue and blocks the signaller thread

58. **Explicit scheduling** occurs when: 1) An accept statement blocks the active task and makes a task read from specified mutex queue 2) A signal moves task from spiecified condition to the signalled stack

59. **Implicit scheduling**: task calls wait() or exit from a mutex member. Monitor will select who next come in. Caller has lowest priority.

60. **Implicit signal monitor** has not condition variables (eg. use waitUntil statement). **Explicit signal monitor** has CV.

61. C: caller. S: signaller. W: signalled.

signal type	priority	No priority (barging, bad)	Note
Blocking	Priority Blocking (Hoare) C < S < W (uC++ signalBlock)	No priority Blocking C = S < W	Priority Blocking wait for cooperation. No priority Blocking requires signaller task Jto recheck condition.

Nonblocking	Priority Nonblocking C < W = S (uC++ signal)	No priority Nonblocking C = W < S (Java/C#)	Priority Nonblocking supply cooperation. No priority Nonblocking
Quasi-blocking	Priority Quasi C < W = S	No priority Quasi C = W = S	makes cooperation too difficult
Immediate Return	Priority Return C < W	No priority Return C = W	(signaller return after signal) are not powerful enough to handle all cases
Implicit Signal	Priority Implicit Signal C < W	No Priority Implicit Signal C = W	good for prototyping but have poor performance

62. **Java Monitor**: Java has **synchronized** class member, and all classes have one implicit condition variable. It allows barging.**Spurious wakeup** => require loop around wait.

63. **Task**: has its own execution state (like coroutine), has a thread of control and begins execution in the task main, and provide mutual exclusion/synchronization (like monitor)

64. **Conditional Accept**: _When (C) _Accept(M), the condition must be true, and call to the specified member must exit. If all _Accept is conditional and the condition is false, it will do nothing. If some condition is true, it will be blocked to wait until the specified member is called. _Else statement will cause busy waiting.

65. **Accept a destructor** : destructor will be executed after main finished. (caller is blocked instead of acceptor)

66. **Increasing Concurrency**: Small overlap between client and server (client gets away earlier) increasing concurrency.

67. An administrator is a server managing multiple clients and worker tasks.

- Makes no call
- Maintain a list of work to pass to worker tasks

- Worker:
    - timer: prompt the admin at specified time intervals
    - notifier: perform a potential blocking wait for an external event
    - simple worker: do work and return the result to the admin
    - complex worker: do work and interact directly with client
    - courier: perform a potentially blocking call on behalf of the admin

68. Two protocol: ticket and call-back routine.

69. **Callback Routine**: when the result is ready, the routine is called by the task generating the result. callback routine cannot block the server. Advantage; server do not need to store the result. client can write the call-back routine(decide pool or block)

70. **Futures:** provides the same asynchrony without an explicit protocol.It is an object that is a subtype of the result type. The future is returned immediately to caller after call without blocking. Callee will fill in the future in the future. When caller tries to use the future, it will either block (until result are ready) or get the result.

71. **_Select**: work with _When and logical operators || and &&. Each _Select-clause action is executed when its sub-selector-expression is satisfied. Control does not continue until the selector-expression associated with the entire statement is satisfied.

72. Optimization:
    1. Reordering: data and code reordered.
    2. Eliding: Remove unnecessary data and computation
    3. Replication: Duplication (code, memory, data) makes processing faster

73. Sequential optimization
    1. Reorder disjoint operations
    2. Elide unnecessary operation

3. Execute in parallel if multiple functional-units.

74. Cache

1. Cache loads 32/64/128 which called cache line.
2. Cache coherence is hardware protocol ensuring update of duplicate data
3. Cache consistency addresses when processor sees update
4. Cache trashing: Too many duplicate cache lines, resulting in excessive cache updates.
5. Because cache line contains multiple variables, cache thrashing can occur inadvertently, called false sharing.

75. Preventing optimization problem

- All optimization problem result from race on shared variable. Two solutions.
- **Ad noc**: Programmer manually augments all data races with pragmas to restrict compiler and hardware optimizations: not portable but optimal
- **Formal**: Language has memory model and mechanisms to abstractly define races in program: portable but baroque and not optimal.
- volatile : get fresh reading   atomic stronger -> prevent eliding and disjoint reordering.

76. Atomic data structure

1. **Compare and swap**: atomic compare and conditional assignment CAA
2. **Lock-free stack**: use CAA atomically top pointer when nodes pushed or popped concurrently.
    a. push(Node &n){ for (;;) { n.next = top; if (CAA (top, n.next, &n)) break;}}
    b. pop() { Node *t; for(;;) {t = top; if (t == NULL) break; if(CAA(top, t, t->next)) break; } return t;}

3. **ABA problem**: thread 1 pop, then context switch, thread 2 pop A pop B and then push A. Back to thread 1, thread 1 get a corrupted stack.

- **Hardware Fix**: use a 64/128bit CAA, associate a counter with header node. Every time perform a push or pop, increment the counter.
- **Software Fix**: each tread maintains a list of nodes (called **hazard pointers**), thread updates its hazard pointers while other threads reading them. thread removes a node by hiding it on private list and periodically scans the hazard lists of other thread for referencing to that node. if no pointers are found, the node can be freed,

77. **Locks** VS **Lock-free**

- lock-free has no locks, cannot contribute to deadlock
- when thread blocks, it is never holding a lock to prevent progress of other threads
- lock-free no panacea, performance unclear
- lock can protect arbitrary complex critical section while lock-free only handle limited set of critical sections
- combine lock and lock-free?

78 **LL (load locked)** and **SC(store conditional)** in MIPS, generalize atomic read/write cycle.

- LL instruction loads value from memory into a register, and sets a **hardware reservation**
- SC instruction stores new value back to original or another memory location. Store is conditional and occurs only if no interrupt exception or write has occurred at LL reservation
- If fails, the register will be set to be 0.
- cannot implement atomic swap as two reservations are necessary

79. **Database Transaction**: optimistically executes change, and either commits changes or rolls back

- SPECULATE: start speculative region and clear zero flag.
- LOCK: MOV instruction indicates location for atomic access. (but not visible to other CPUs)
- COMMIT: end speculative region. If no conflict, make MOVs visible to other CPUs. if conflict, set zero flag, discard reservations and restore register back (roll back)

80. **Concurrency Models**

- **Actors**: an admin, accepting messages, handle it or sends it to a worker(fixed/open pool)
    - communication is via an untyped queue of messages
    - Mailbox is polymorphic in message types => dynamic type-checking
    - Message send is usually asynchronous
    - popular in Erlang and Scala
- **Linda**: based on multiset **tuple space**, duplicates are allowed, accessed associatively by threads for synchronization and communication
    - tuple space is unnamed
    - threads created through tuple space, communicate by add read and remove
    - tuple accessed atomically and by content
- **OpenMP**
    - Shared memory, implicit thread management, 1-to-1 threading model (kernel threads), some explicit locking
    - Communicate with compiler with #pragma directives
    - fork/join model

    - fork: initial thread creates a term of parallel threads
    - each thread executes the statements in the region construct
    - join: when team threads complete, synchronize and terminate, except initial which continues

81. Concurrency Language

- **Ada**:
    - **when** clause for external scheduling (only at start of a entry routine), no parameter or local variables allowed
    - **select** is external scheduling (only appear in task main)
    - **no internal scheduling (no condition variables)**
    - **requeue** to make a blocking call, re-blocked at that member's entry queue.
- **SR/Concurrent C++**
    - have **external scheduling** using an **accept** statement. Use **when** to filter calls by reference caller's arguments
    - **by** clause specified the order of entry
    - no internal scheduling
- **Java**
    - **Thread** is like uC++ uBaseTask
    - Thread starts in member **run** b explicit calling **start** after thread's declaration, **termination synchronization** is accomplished by calling **join**.
    - returning result on termination is using task's global variables
    - become object after thread terminates (like uC++)
    - possible to have **public synchronized members** of a task.
    - **no mechanism to manage direct calls** (no accept statement)

- **Go**
    - Non-object-oriented, light-weight, non-preemptive thread. (**goroutine**)
    - **go statement** (like fork/start) creates new user thread running in routine.
    - **cannot reference goroutine object => no direct communication**
    - All thread terminate **silently** when program terminates.
    - threads synchronize/communicate via **channel** (a type of shared bounded buffer)
    - **<-** performs send/receive eg. send: ch1 <- 1, receive: s <- ch2
- C++11 Concurrency
    - thread(Fn &&fn, Args && … args); any entity that is callable (functor) may be started
    - Thread starts implicit at point of declaration
    - use **detach** to run thread independently
    - **It is an error to deallocate thread object before join or detach.**
    - Scheduling is no-priority nonblocking => barging => must use busy wait

82. Libraries

- Java.util.concurrent
    - concurrent aware and memory-model
    - Scheduling is no-priority nonblocking => barging => need busy wait
    - No connection with implicit condition variable of an object
- Pthreads
    - All C library approach have type-unsafe communication with tasks
    - No external scheduling
    - Internal scheduling is no-priority nonblocking => barging => need busy wait