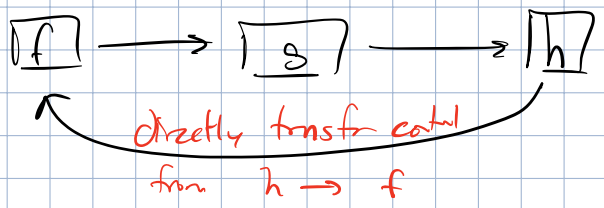
Non-Local Transfer



In single func., multi-level exits are easy → goto label
↳ Problem: labels are scoped to function

Another problem to solve: multiple possible return results!

Traditional Approaches to Multiple Returns

① Return code: indicates normal/exceptional return
↳ printf() → 0 ⇒ good, -1 ⇒ exception

② Status flags: global variables that indicates exception/normal

③ Fixup: pass fixup func. to call when exception returns

④ Return union: combine code w/ object

Go { ok (status), val := foo() (val)
↳ requires checking 'ok' before 'val'

Drawbacks:
1. Checks can be optional
2. Mixing return w/ exceptions
3. Multiple exception types ⇒ bloat in return types/codes
4. Overwrite codes/statuses
5. Increasing param # w/ fixup

Dynamic Multi-level Exit

① Set a global label var

② B/f func call, set label var

③ Throw exception → jump to global label → 1. Direct control flow to label (updating PC)
2. Go to transfer point

④ On any exit/return/exception handling, reset to prev label

Skips stack scopes! Will clean up stacks implicitly → pseudo multi-level exit

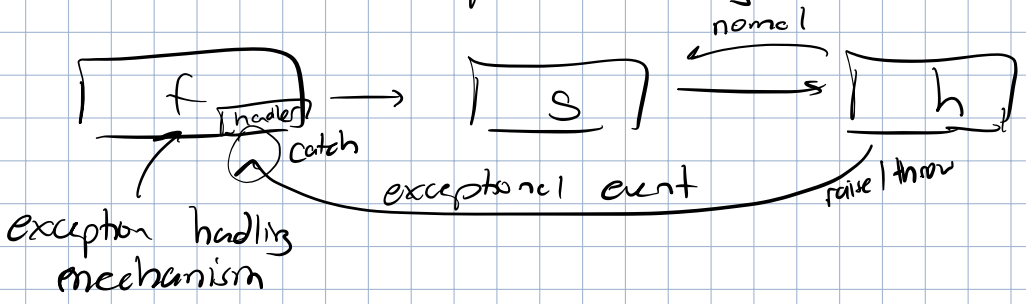
Dynamic: you can never know what global label is set to from reading code!

In C/C++: jmp_buf → label
setjmp → sets global label
longjmp → goto label

Problem: too general

Exception Handling

Non-local transfer ≡ exception handling



Types of execution:

① Source: execution (program w/ own stack) that raises an exception

② Faulting: " changes control flow (eg. whatever catches)

If ① = ② ⇒ local exception. O.W.: non local exception.

Non-local exceptions can be concurrent, or not!

Propagation: direct exception from source → faulting

↳ Order of catches is most recent func. w/ corr. catch.

Once exception reaches handler:

① Return → handler how handled exception

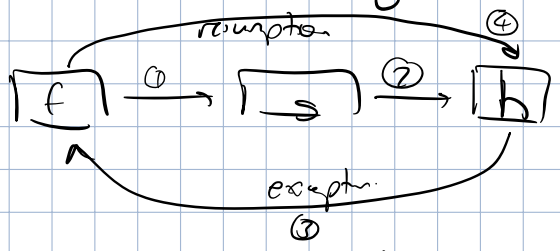
② Reraise (often to add extra info)

③ Raise a new (private exception)

In C++:

```
guarded block { try {
                  ...        ← code here can throw an exception.
                } catch (exceptionType) {
                  ...
                }
```

Resumption: Control flow goes back to raise



Q: What happens to stack / objects if exc. thrown?

↳ Terminate block: call dtors & code in 'finally'

```
try {
  - -
} catch () { - - }
} finally { - - }
```

← will always be exec.

Note: we don't directly jump to routine that handles exception → stack unwinding must happen.

Implementation

Approach ①: Use labels

1. Assoc. global label / exception type
2. Set label on guarded block entry
3. Reset global label of catch / exit of block

Con: lot of ops. if lots of try & catches. Exceptions not raised too often

Approach ②: Lookup table

① Store catch & other info per block. } amortize costs on
② Exc. raised → do linear walk in LTE } exception raising?

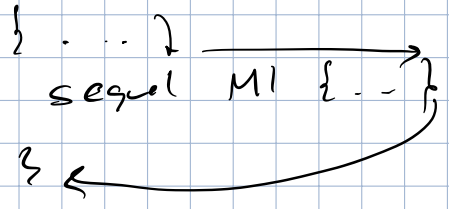
Static / Dynamic Call / Return

- Static / dynamic call: routine name is known @ compile-time (static) or not (dynamic)
- Static / dynamic return: routine ret. location known @ call

return	static	dynamic
static	sequel	termination exception
dynamic	reg routine	passing routine into func. / resumption

Sequel: routine w/ no return value.

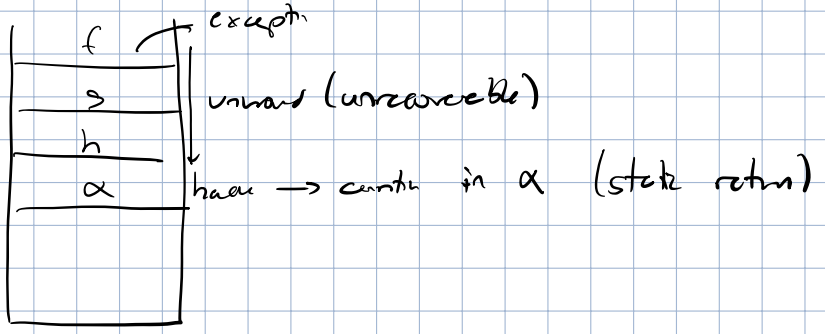
↳ Control returns to end of block where sequel is declared



* Often used for non-recoverable event (eg. FileException, Stack Overflow) *

Not good for modularize

Termination Exception: dynamic raise (handler is unknown), static return b/c we end handler ← stack is unwound.



In C++:

```
foo (){
  ...
  throw E()
}
```

⟹

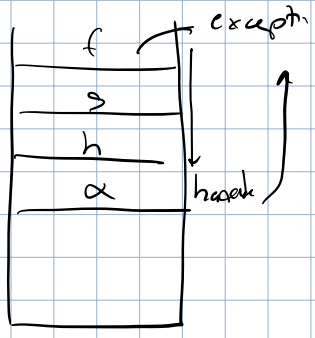
```
bar () {
  ...
  try {
    ... foo() ...
  } catch (E) {...} catch ellipsis
```

specify type of try

Key note: if you don't want to handle throws → no need to make catch ← goes down stack

try ... retry: restart try block after exception handled

Resumption: dynamic raise b/c handler is unknown but stack is not unwound. At end of handler, don't know where you will resume → dynamic return.

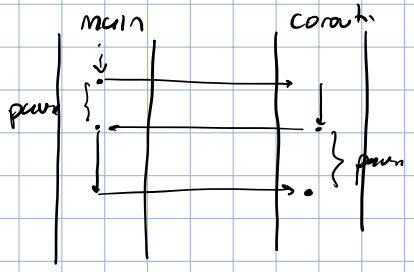


⟹ Corrective action

Coroutines

Defn: routine that can be paused & resumed.
↳ Why? Has it's own stack! Save state.

Not concurrency ⟸ not doing simultaneous comp.



Semi-coroutines

Structure:

```
- Coroutine <name> {
    private members: ⟵ comm.
    void main() {
        ---          ⇒ suspend in here
    }
  public:
    var foo() { .. resume() .. }
```

1st resume: starts new stack for coroutine.

Suspend: reactivates last resume() caller

Coroutines can be deleted b/f done

Since suspend() will save states on stack → no need for flag variable!

Error handling

① Defining exceptions:

```
- Coroutine <...> {
    private:
      ---
    public:
      - Event <ExceptionName> {..}   ✓ class exception
      .
      .
  }
```

② Raising exceptions:

Allows non-local exception passing!

- Throw ExceptionName(); OR - Resume ExceptionName() [-At coroutine]
  ↳ Go with this

In μ++: exception caught by lowest level class, not parent class

③ Handle exception:

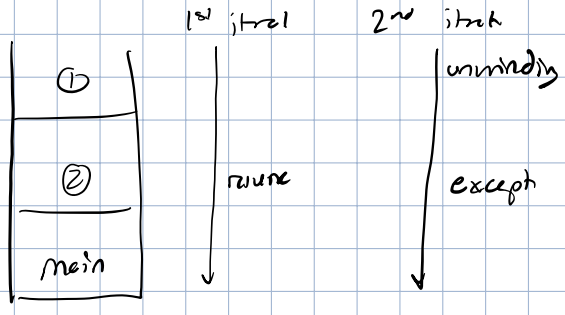
```
void main () {
  ...
  try {
    _Enable {
      ...
    }
  } _CatchResume (...) {
    ...
  } catch (...) {
    ...
  }
}
```

Allows exceptions resumed/thrown at routine to be handled

Resumption: no stack unwinding

Exception: stack unwinding

Q: what happen if _Resume but no _CatchResume?



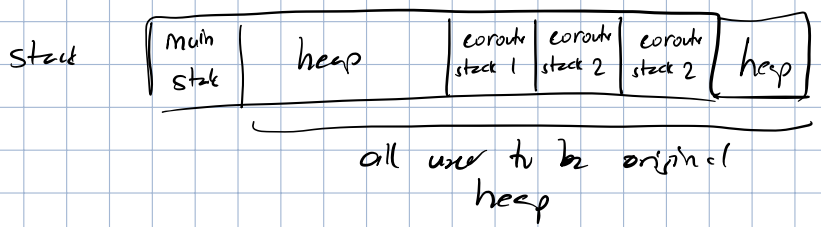
Multiple exceptions at coroutine → queued & in FIFO

Can always disable exception handling via _Disable {...}

Starter: prog. that does first resume

↳ Semicoroutine: only starter calls resume (not multiple coroutines calling resume())

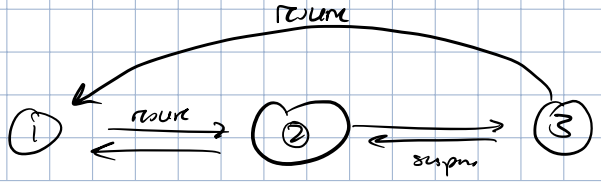
Memory management



Stacks small in coroutine → don't allocate big objects

Full coroutines

Resume cycle:



resume():
① Pause curr. routine (uThisCoroutine())
② Activate curr. object

vs.

suspend():
① Pause curr coroutine (uThisCoroutine())
② Activate last resumer

Constructing full coroutine:

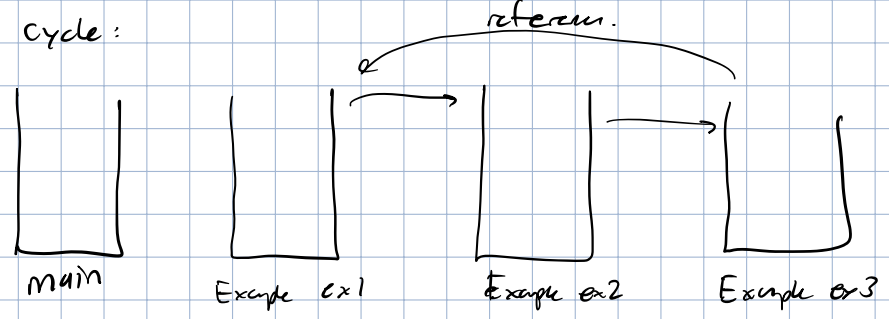
A: Coroutines need reference to each other:

```
- Coroutine Example {
    Example next;   <- reference
  private:
    void main {
      cout << "ID: " << id << endl;
      next.call()
    }
  public:
    Example (Example next) : next(next) {};
    void call {
      resume();
    }
}
```

set reference

If cycle of references, might need special function to make reference

B: Start cycle:



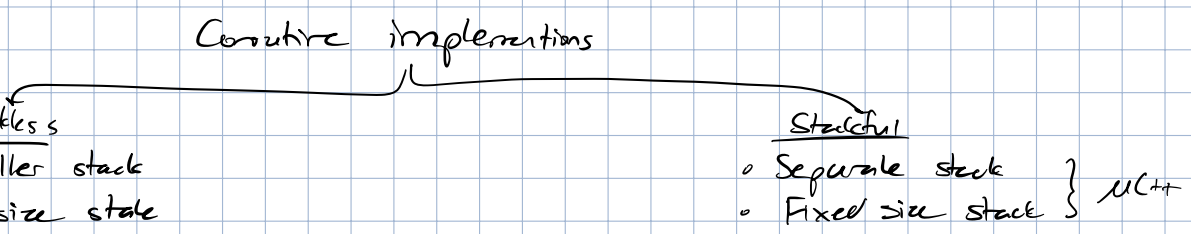
1. main calls ex1.call():
   resume() → inactivate main
   activate "this" → ex1
2. "ID: 1"
3. call ex2.call()
   resume() → inactive ex1
   activate "this" → ex2
4. "ID: 2"
5. call ex3.call()
6. "ID: 3"
   ⋮

C: End cycle

Get back to starter, but not always the last resumer

Ending coroutine → starter

Coroutine in modern languages



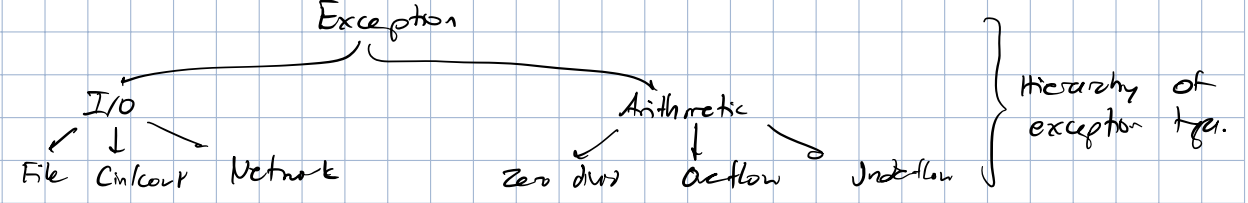
- Python: yield (suspend) & next (resume)

JS: yield & ... .next()

C++: has API but not good

More Exceptions

Derived Exception Type



Hierarchy enables programmer to handle exceptions at diff. granularities
↳ Less coupling to specific exceptions

To handle specific & general case:

```
try {
  ...
} catch (SpecificExc. & e) {
  ...
} catch (GeneralExc. & e) {
  ...
}
```

Note: catch by reference ⇒ dynamic casting, re-resumes

Catch - Any

Purpose: catch any sort of exception

Soln:

① Catch base exception class

② Syntax:

C++: `} ... catch (...) { .. }` ⇒ Applies to _CatchResume

③ Finally (in Java)

`try { .. } catch (..) { .. } finally { .. }` ↙ Always executed

Exception Parameters

Goal: pass info from raise point → handler

Soln:

1. Define struct for exception
2. At raise: initialize params
3. Catch by reference → use info
   - For resumption ⇒ change info in exception → _Resume

Exception List

Goal: specify which routines are possibly throwing an exception ⇒ static checking

- Ex: // `int f() throw (E, B, ...) { ... }`

Java uses a static check, C++ uses dynamic runtime checks

Issue w/ exception lists:

1. Cannot know all exceptions from routine
2. Fixed set of exception types

Destructors

Most times, dtors will not throw exception. Can force it to throw exceptions

- `class A {`

  `~A() noexcept(false) { .. }`

Multiple Exceptions

Not usually an issue unless dtor raises exception → undefined behavior

Multiple exceptions can be thrown, but only 1 can propagate

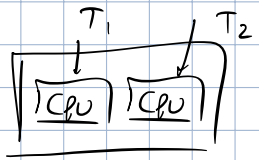
CONCURRENCY

Thread: indep. sequential exec. of prog w/ own state

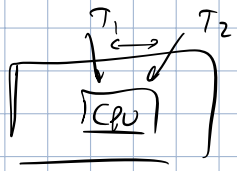
Process: has own thread & state into (like coroutine). Can have multiple thread

Task: light-weight process, smaller & lighter

Parallel execution:



Concurrent execution:



switch so fast that it looks like parallel execution

Why?

① Problem divided is easier to solve

② Speed ups

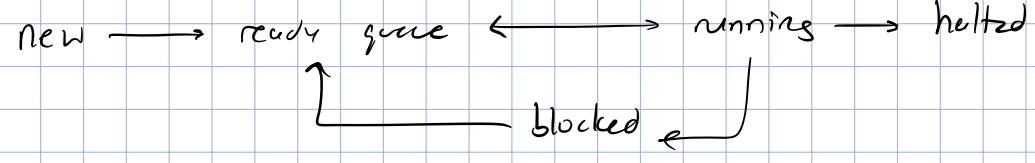
Difficulty?

① Understanding coordination, simultaneous exec.

② Problem specification: how to break up problems, comm.

③ Debug: non-deterministic order

Execution States



Scheduler determines who should run → cannot skip ready queue

Operations that are non-atomic can switch from running state midway

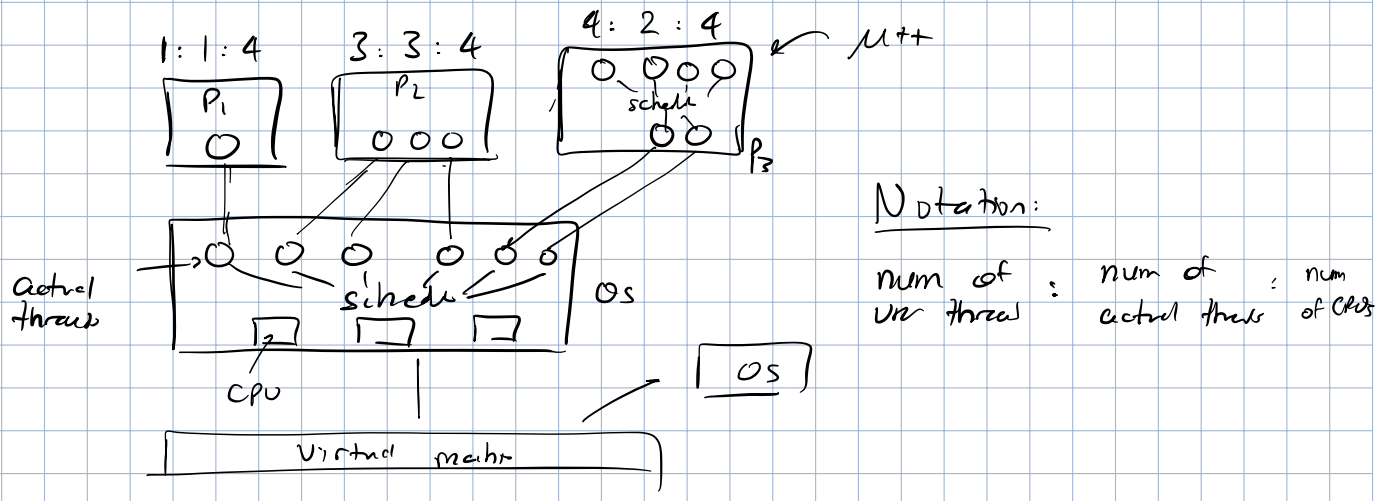
↳ Can lead to data loss

Ex:// 3 threads running:

```
int global = 0
for (int i = 0; i < N; i++) {
    global += i
}
```

↳ Threads can switch halfway

Threading model



Notation:

num of VM threads : num of actual threads : num of CPUs

Concurrent Systems

Types of concurrent systems:

① Automatically converts sequential → concurrent prog.

② Provide hints/annotation to sequential → compiler converts to concurrent } Implicit

③ Explicit concurrency constructs

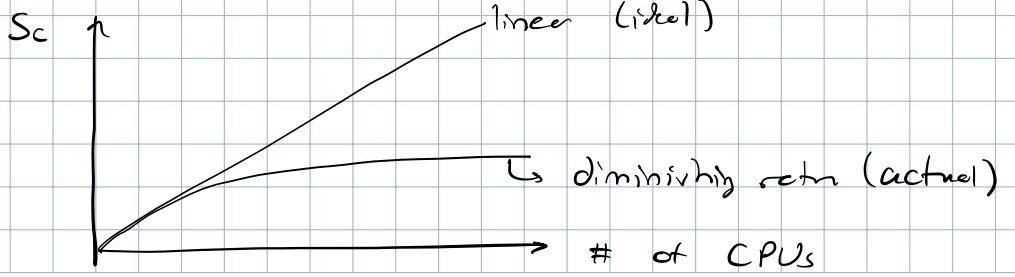
At some point, you need to use explicit concurrency

Speedup

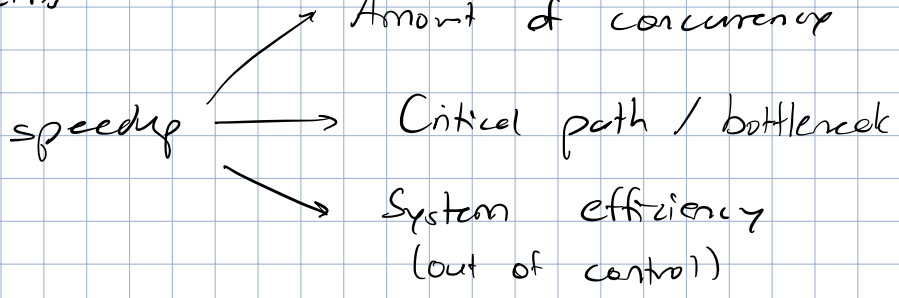
$$S_c = \frac{T_1}{T_c}$$

T_1 ← time in seq. prog

T_c ← time in concr. prog.



Speedup factors:



Amdahl's Law:

Speedup of C CPUs w/ prog taking P% time in concurrency is:

$$S_C = \frac{1}{\underbrace{1-P}_{\text{seq.}} + P/C} \quad \text{(assuming } T_1 = 1,\ T_C = \text{seq} + \text{concr.)}$$

Ex:// Seq. takes 10 seconds on 1 CPU. 2.5 sec for 4 CPUs w/ 100% concurrency

$$S_4 = \frac{1}{1-1 + \frac{1}{4}} = \frac{4}{}$$

↑ P = 100%.

Ex:// 80% concurrency:

$$S_4 = \frac{1}{1-0.8 + \frac{0.8}{4}}$$

$$= \frac{1}{0.2 + 0.2}$$

= 2.5 times speedup } → Huge decrease b/c of sequential code

To get real speedup → minimize time in sequential prog. & reduce bottlenecks

Thread Creation

Any concurrency construct requires:

1. Creation
2. Synchronization
3. Communication

① COBEGIN/COEND

```
# include <@Cobegin.h>
```

```
int main (..) {
  :
  COBEGIN
    BEGIN ... END  → Each BEGIN in theory has own
    BEGIN ... END    thread
    BEGIN ... END
  COEND
    └→ Wait for all threads to finish
}
```

Implicit concurrency

```
COFOR (i, start, end, {..})
Runs threads indexed start
→ end-1 that runs {..},
can access thread idx using i
```

② START/WAIT

```
#include <uCobegin.h>
int main (..) {
  ..
  auto ptr1 = START (func, params)   → Thread starts running func w/ params
  auto ptr2 = START (..)
  :
  i = WAIT (ptr1)   → Wait until thread finishes & return the value of func
}
```

Explicit concurrency

START/WAIT can simulate COBEGIN/COEND, not v.v.

③ Thread object

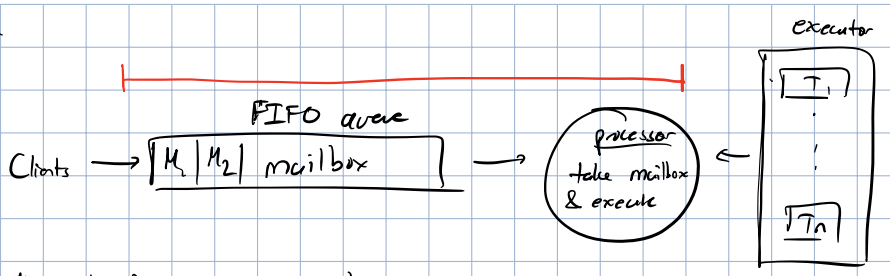
```
_Task <name> {
  void main () {..}   ← Function runs if thread started
  :
};
```

Initialize:

```
{   ← COBEGIN
  :
  <name> obj ;        OR    <name> * obj = new <name>   ← START
  :                         :
}   ← COEND                 delete obj ;   ← WAIT
```

Explicit concurrency!

④ Actor




```
#include <uActor.h>

struct StrMsg : public uActor::Message {
  ...
    StrMsg(...) : Message (uActor::Delete /NoDelete) {...}
};

_Actor <Actor Name> {
    Allocation receive (Message & msg) {
        Case (StrMsg, msg) {
            msg_d →  StrMsg
        } Case (BoolMsg, msg) {
            msg_d → ...
                BadMsg
        } ... {
            return Delete
        }
        return NoDelete
    }

int main (...) {
    uActor::start();
    * new <Actor Name> (*new StrMsg(...)|...|...|uActor::stopMsg
    :
    uActor::stop();
}
```

delete after message consum
No delete after use

dynamically cast message of type checks

Figure out type of msg from derived msg struc.

Action to take for Actor

default msg

set threads ready for actor

will wait until all uActor::stop() called

Implicit concurrency

Actions of _Actor finish:

① Run destructor

② Delete memory

③ Update storage into and remove from actor sys.

Allocation types:

Nodelete ⇒ do no ①, ② or ③

Delete ⇒ ①, ②, ③ (Use for dynamic alloc.)

Destroy ⇒ do ①, ③

Finished ⇒ do ③. Use for stack allocs

Exceptions

Unhandled exceptions go to last joiner

Same syntax, but faulting exec. not halted as soon as exec. arises. Check at

① _Enable begin & end

② suspend / resume

③ yield call

④ _Accept call

Syncing & Comms During Execution

Synchronization: thread waits for another thread to reach some point

↳ If sync'ed & both threads ready to recv & transmit info → comm.

Busy waiting: thread in loop until event happens

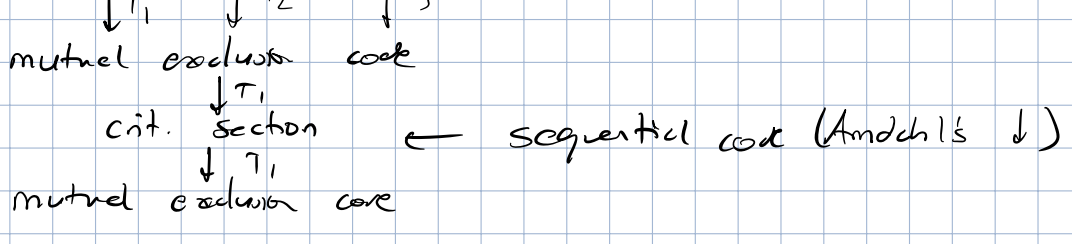
Comm.: changing shared memory (by reference), channels, ...

Critical section

Obj: want to access an object one thread at a time

↳ Note: thread can be interrupted on obj. access, but no other thread can change @ same time

Ans: mutual exclusion code



← sequential code (Amdahl's ↓)

Beware: some objects can have multiple threads reading an object ⇒ keep read concurrent

Note: static variables are shared in all object instances → not safe

Principles of mutal exclusion

Q: Can we write code that forces serial execution?

A: Yes!

Rules that we <u>try</u> to follow:

1. Safety: 1 thread access C.S. @ a time
2. Eventual progress: each thread will eventually execute & can run in any order
3. No selfishness: thread not in entry/exit cannot prevent other code from accessing C.S.
4. No livelock: cannot postpone selection of thread to enter C.S. } Relate by busy waiting
5. No starvation: thread must eventually enter C.S. }

Mutual exclusion ⇏ FIFO service (there is unfairness)

↳ Barging: waiting thread overtaken by arriving thread

Software Solutions

① Dekke's Algorithm

```
for (;;) {
    me = WantIn
    if (you == DontWantIn) break
    if (Last == me) {   ⇒ Forces alternation if want to go in again
        me = DontWantIn
        while (Last == me && you == WantIn) {}
    }
}
CriticalSection()
if (Last != me) {
    Last = me
}
me = DontWantIn
```

Issue: not R/W safe won't highlighted code (Hesselink)

↳ Not all R & W are atomic. R/W safe → works for non-atomic R/W

Failure Case:

T_0	T_1
last = me	
me = DontWantIn	
(flicker DontWantIn)	you == DontWantIn → true
	CriticalSection
	last = Me
flicker WantIn	
	you == DontWantIn → false
	wait ↓
flicker DontWantIn	
terminate.	

Unbounded overtaking: thread can overtake multiple (unbounded #) times to execute
↳ Thread can come back & reenter C.S. This is because loser retracts intent

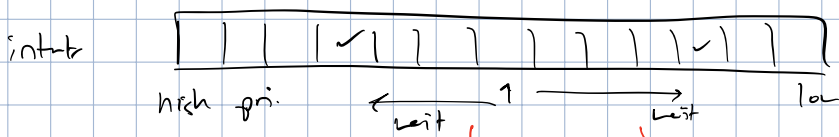
② Peterson's Algorithm

```
me = WantIn
Last = me
while (you == WantIn && Least = me) {}
critical section ()
me = DontWantIn
```

R/W unsafe b/c it needs atomic read & write

Bounded overtaking: loser of Last = Me still wants in (doesn't retract intent)
↳ Loser cannot come back & re-enter C.S.

③ N-thread prioritized entry




```
Do {
  intents[priority] = WantIn
  for (lower indices j) {
    if intents[j] == WantIn
      intents[priority] = DontWantIn
      while (intents[j] == WantIn) {}
      break;
  }
} while (intents[priority] == DontWantIn)

for (higher indices j)
  if (intents[j] == WantIn) {}
}
```

Check if hi pri. thread has placed intent → retract & wait

Check lo pri. If want in → wait

criticalsection ()
intents [priority] = Don't Want In

Can have starvation, only need N bits (N = # of threads)
↳ Can do 3-bit, but R/W set. 4 bit soln. ⇒ R/W safe.

④ N-thread Bakery

ticket values ∈ (0, ∞). Lower tickets ⇒ higher priority
↳ 0: thread is selecting. ∞: don't want in

Algo:

```
ticket [priority] = 0          ← prevent other threads from entering
int max = 0
get_max_ticket in ticket arr
ticket [priority] = max + 1
for (int j : 0 → n) {
    while ticket [j] < max || (ticket [j] == max && j < priority) {}
}
criticalsection ()
ticket [priority] = INT_MAX
```

Ticket selection

Tiebreaker

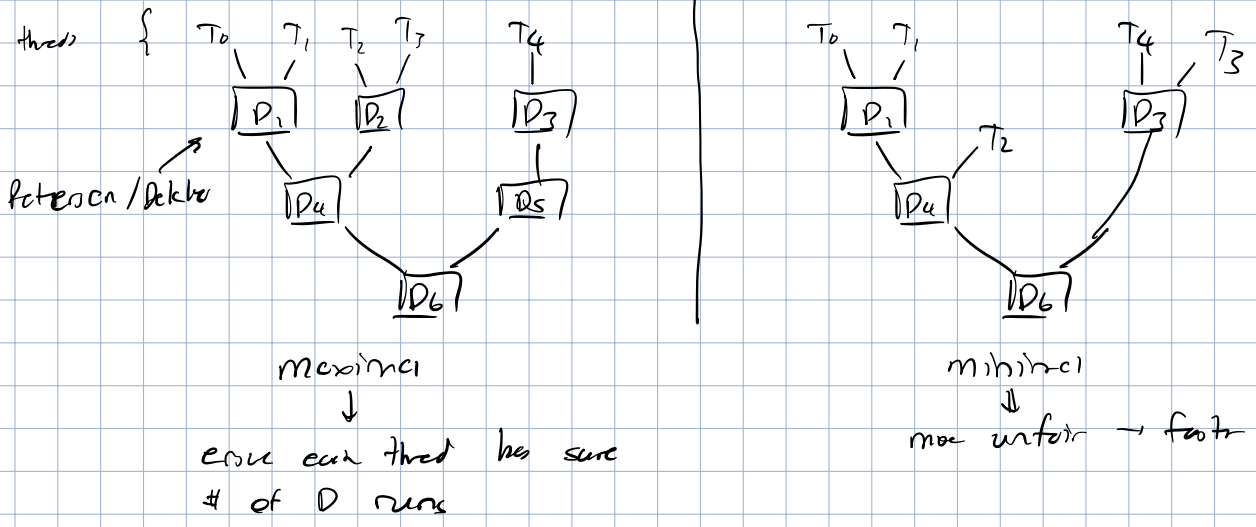
Wait for turn.

Space: NM bits
(N = # of threads, M = ticket size)

No starvation

R/W unsafe

⑤ Tournament



maximal
↓
ensure each thread has same # of D runs

minimal
↓
more unfair → faster

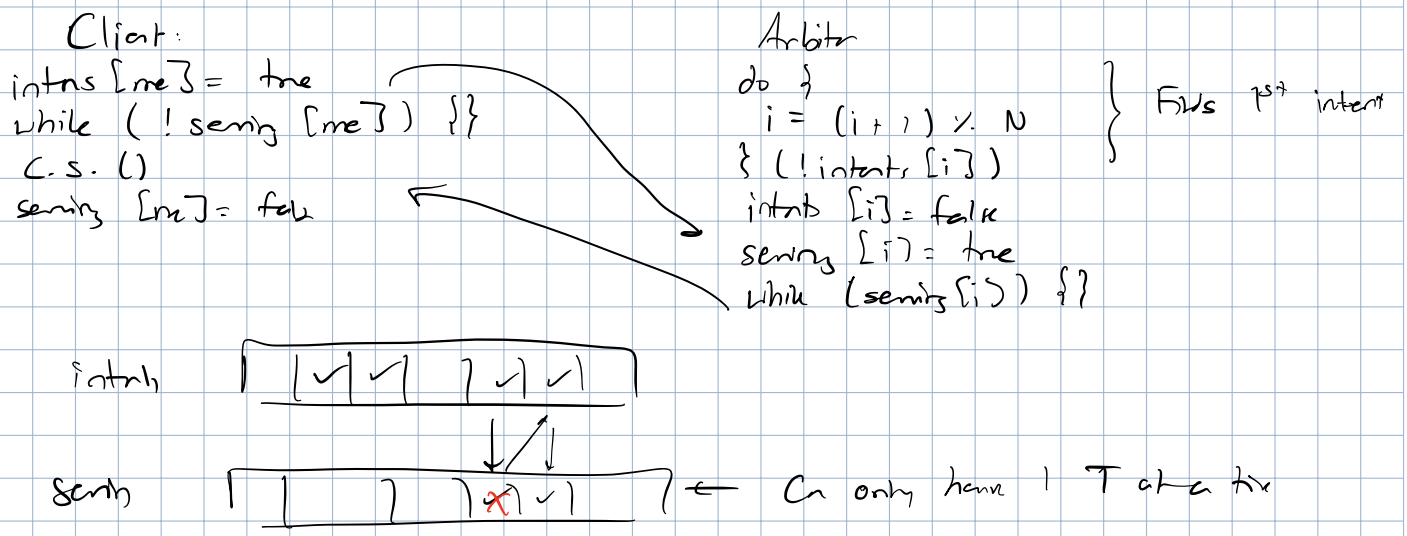
Intents for threads must reverse bottom up

Properties of D_i → entire tree has property if all nodes implement it

All have unbounded overtaking

Size (minimal) = $\underbrace{(N-1)}_{\#\text{ of true nodes}}$ M bits — node size

⑥ Arbiter



Mutual exclusion → synchronization

No starvation

R/W unsafe

Con: arbiter thread overhead.

Hardware solutions

SW solutions are complex → use hardware.

HW provides atomic read & write ops. → cheats!

Atomic instr.:

① Test/Set instr

```
int TestSet (int & b) {
    int temp = b
    b = CLOSED
    return temp
}
```

→

```
Code:
While (TestSet (Lock) == Closed) {}
C.S.
Lock = Open
```

Breaks rule 5 (starvation possible)

② Swap

```
void Swap (int & a , int & b) {
  int temp;
  temp = a
  a = b
  b = temp
}
```

→

```
Code:
int dummy=Close , lock = Open
do {
  Swap (Lock , dummy)
} while (dummy == close)
C.S.
lock = open;
```

③ Fetch & increment

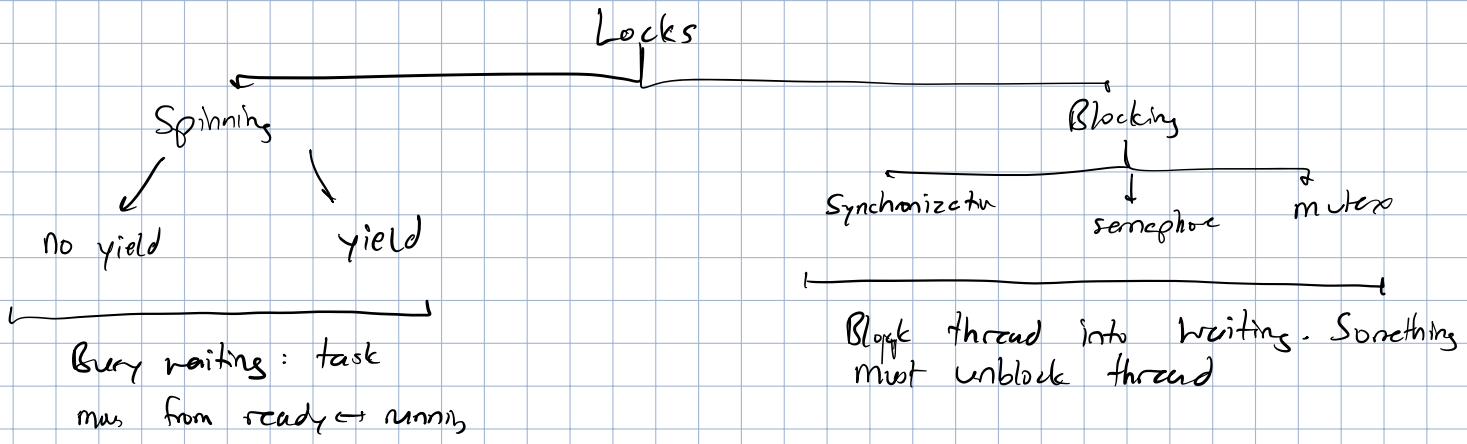
```
int FetchIn (int & val) {
  temp = val
  val += 1
  retn temp;
}
```

→

```
Cod:
While (FetchIn (Lock) != 0) {}
  C.S.
Lock = 0
```

Can overflow, but easy to resolve.

Locks



Spin Locks

Via HW ops : While (TestSet (Lock) == Closed) {}

Issue : wasteful of CPU cycles

↳ Optimize : yield during spin or move back thread to ready state

Breaks rule 5 → starvation possible

Necessary if computer has nothing else to do

Implementation in μ++ :

① uSpinLock : non-yielding
② uLock : yields

} lk.acquire();
  ⋮
  lk.release();

Ex:// Synchronization

```
_Task T1 {
    uLock &lk;
    void main() {
        S1
        lk.release();
    }
    ⋮
```

```
_Task T2 {
    uLock &lk;
    void main();
        lk.acquire()
        S2
    }
```

Ex:// Multiple locks:

```
_Task T1 {
    uLock &lk1;
    uLock &lk2;
    void main() {
        lk1.acquire();
        S1
        lk1.release();
        lk2.acquire();
        S2
        lk2.release();
    }
    ⋮
```

→ 2 sections can be run @ same time, but 1 thread / crit. section

# of locks ≤ # of critical sections

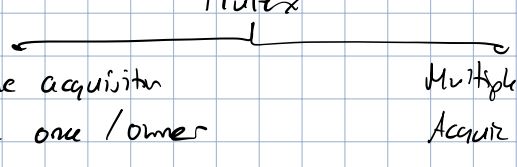
Blocking Locks

Blocking lock needs semaphore to release blocked threads → less work, more coordination

All blocking locks have:
① State to facilitate lock semantics
② List of blocked acquirers } → any one can be chosen to proceed

A: Mutex lock

Obj: mutual exclusion



Implementation:

```
class MutexClass {
    uLock spinlock;
    Task* owner;
    bool avail;
public:
    void acquire() {
        lock.acquire();
        while (!avail && owner != currThread) {
            yield(&lock);
            lock.acquire();
        }
        avail = False;
        C.S.
        lock.release();
    }
```

Barging: thread can come in b/t lock acquire

```
void release() {
    lock.acquire();
    owner = nullptr;
    if (!blocked.empty()) {
        remove task & make current
    }
    avail = true;
    lock.release();
}
```

2 methods of dealing w/ barging:

① Avoidance

```
void acquire() {
    lock.acquire();
    if (!avail && owner != currthread) {
        yield(&lock);
    } else {
        avail = False;
        lock.release();
    }
    owner = currThread;
    lock.release();
}
```

barged set blocked

```
void release() {
    lock.acquire();
    owner = nullptr;
    if (!blocked.empty()) {
        make task ready
    } else {
        avail = true
    }
    lock.release
}
```

passes control to blocked thread

② Prevention

```
void acquire() {
    lock.acquire();
    if (!avail && owner != currThread) {
        yield(lock);
    }
    avail = true;
    owner = currThread();
    lock.release();
}
```

```
void release() {
    lock.acquire();
    owner = nullptr;
    if (!blocked.empty()) {
        remove task & make ready
    } else {
        avail = True
        lock.release();
    }
}
```

μ++ implementation (multi-acquire): uOwnerLock();
↳ Must call release() == # of times acquire() called.

Common patterns:

① - Finally:

```
lock.acquire();
try {
  ...
} - Finally {
  lock.release();
}
```

② RAII

On ctor: acquire lock
On dtor: release lock

Stream locks: osacquire(stren) << "..." << endl; ↑ release lock on stream

B. Synchronization Locks

Also known as cond. lock.

Special note: signals to release can be lost if no task present & always block on acquire

Suffer from some barging problem

① Avoidance:

```
milk.acquire();
if (!slk.empty()) { s.release(); }
else { occupied = false; }
milk.release();
```

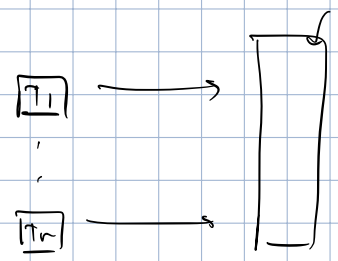
② Prevention

```
milk.acquire();
if (!s.empty()) s.release();
else { occupied = false; milk.relase(); }
```

μ++ class: uCondLock: wait, signal, broadcast
↓ block, ↓ wake up 1 task, ↓ wake up all tasks

Need mutual exclusion lock to prevent signal from being lost

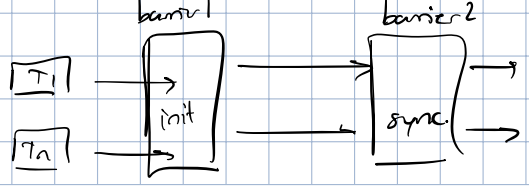
C: Barrier



Barrier forces threads to wait until all thread good
↓
sync.

Last task unblocks all tasks on barrier

Another common use:



μ++ impl. : - Cormonitor

- Cormonitor <Name> : public uBarrier {

  protected:

    void last() ← func. executr when last thread entes barrier.

  public

    void block() ← func. executr if threar calls block()

}

implicitly mutually exclusive

Cannot use this in COFOR: implicit concurrency → don't know how many threads will be created & need to be in barrier.

Q: Semaphore

Equivalent to yielding spin-lock, but blocks instead

Semaphore

- Binary: 0 & 1
- Counting: Multiple states

Common:

- .P(): decrements semaphor value & waits if value is 0
- .V(): increment semaphor value & unblocks waiting task. (usually longest waiting)

① Binary:

Mutual exclusion:

initial state

```
uSemaphor mlk(1);
  :
mlk.P();
  C.S.
mlk.V();
  :
```

Synchronization:

```
uSemaphor lk(0);
void f() {
  S1
  lk.V();
}
```

```
void g() {
  lk.P();
  S2
}
```

② Counting semaphores

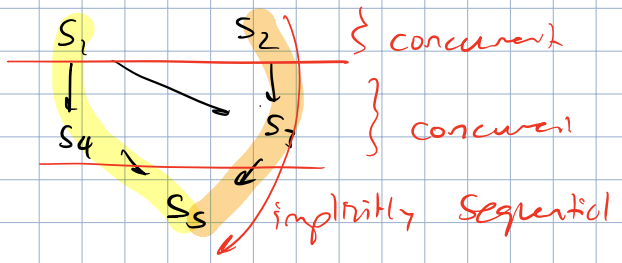
Multiple threads enter critical section → places limit

Lock Programming

Precedence Graph

COBEGIN + semaphore is as powerful as START/WAIT

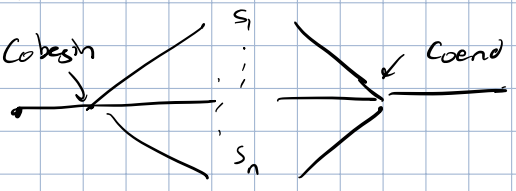
Ex:// Code up precedence graph (dependency graph)



Sol'n:

```
vSemaphore L1(0), L2(0);
COBEGIN
   BEGIN  S1; V(L1); S4; V(L2); END
   BEGIN  S2; P(L1); S3; P(L2); S5; END
COEND
```

* Note: precedence graph ≠ process graph! Process graph will always look like letter:



Tip: location of P() & V() matters! Only say V() after all state is changed

Split binary semaphores

Defn: collection of binary semaphores whose sum ≤ 1

Use: different types of tasks how to block separately & need baton-passing

↳ Baton passing: pass off mutual exclusion to one type of task. (barging prevention)

Also avoid temporal barging (tasks acquiring resource b/f they should).

↳ Sol'n: use a queue or private semaphores (each task has own semphr)

Indirect Communication

Problem: semaphores require complex prog. We want higher-level concurrency constructs

Critical Regions

Idea: declare variables as shared & guarantee M.E. when changing variables.

```
VAR <variables> : SHARED <type>   ← Shared variables that compiler will guarantee M.E.
  :
  ;
REGION <variable> DO
  :
  !
END REGION
```

} Provides M.E. for variable.

Problems:

① You can still read while writing → problems.

② Nested critical regions → deadlock

Conditional region:

```
REGION <variable> DO
  AWAIT conditional-expr.   ← F (busy-waiting)
  :
  ;
END REGION                  ← T
```

Monitor

```
_Monitor <class_name> {
  shared variables          ⇒ Not guaranteed to have M.E.
  public:
    member1();
      ;
    _Nomutex member2();     ⇒ Not M.E. method
}
```

} Public methods guarantee M.E. unless specified.

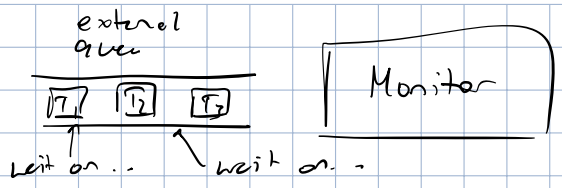
Recursive entries allow (multiple acq. lock)

Dtor releases implicit mutex, so thread will block if in Monitor

Q: How to schedule tasks & provide synchronization?

① External scheduling.

```
void method() {
  <cond> _Accept( method1 );   ⇒ If <cond> true, threads will block until someone finishes method
  :
}
```



② Internal scheduling

Use condition variables: uCondition <variable>; <variables [..];

↳ Queue of waiting tasks! Like cond lock

Usage:

```
_Monitor Example {
    uCondition a;
    public:
    void method1() {
        a.wait()
        ;
        a.signal()
    }
}
```

→ Atomically places thread at back of a's queue.

→ Remove front of queue & set it ready
↳ Not allowing signaller to run until signaller finishes method

signalBlock(): signals thread & blocks signaller

*Important: guarantees no barging! *

When to use external vs. internal?

External should be used if:

A: scheduler depend on param.
B: need to block but cannot guarantee signalling

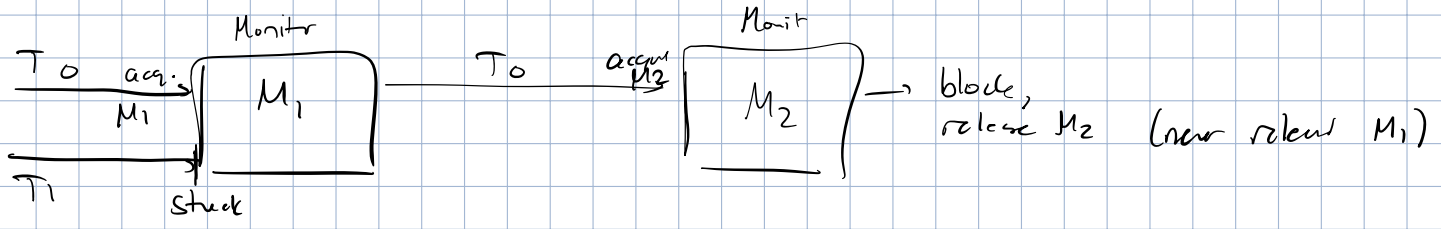
Monitor exceptions

```
void mem1() {
    ...
    _Throw E();
}

void mem2() {
    try {
        _Accept(mel);
    } catch (uMutexFailure::RendezvousFailure &) {. . }
```

Exception type of _Accept method

Nested monitor



Solution: nest M_2 defn. in monitor M_1

```
_Monitor M1 {
    _Monitor M2 {
        ...
    }
public:
    ...
};
```

Intrusive lists

Allocating on heap is a huge bottleneck

↳ Soln: allocate on global stack/heap w/out copying overhead

μC++ intrusive lists either:

① uColable: 1-way linked list
② uSeqable: 2-way linked list

} Have their own iterators.

These intrusive D.S. auto dealloc @ end of scope → no dynamic alloc necessary.

Counting semaphore vs. Monitor

Monitors can simulate semaphore

```
_Monitor Semaphore {
    int sem;
    uCondition semCond;
public:
    Semaphore(int cnt=1) : sem(cnt) {}
    void P() {
        if (sem == 0) semCond.wait();
        sem--;
    }
    void V() {
        sem++; semCond.signal(); }
```

Monitor types

Monitor

Implicit scheduling

Task waits/exit & monitor figures out who should proceed when monitor is empty

Either: calling (C) stack,
signalled (W) stack,
signalle (S) stack ← signal Block

Explicit scheduling

Internal schedulin
- Accept

Extenal schd
- wait / . signal

Programmer writes code to schedule & synchronize

We classify monitors based on implicit scheduling priorities

Implicit signalling:

Monitor that waits on predicate & wakes up all waiting threads on mutex membr exit to chck if pred is T → proced.

```
_Monitor A {
  public:
    void mem() {
      waitUntil <pred>;
      ...
    }
    void foo() {
      ...
    }
};
```

If thread completes:
① Wake up all waiting threads
② Evaluate <pred>
③ Continue thrd if <pred> == T

Immediate-return signal

```
_Monitor A {
  public:
    void foo() {
      ...
      cond.signal();
    }
};
```

↓ Immediately returns, no blocking

Powerful but not powerful enough for all cases! May need to do more work after signal()

Monitor type chart

signal type	priority (no barging)	no priority (barging)
blocking	$C < S < W$ (signal Block)	$C = S < W$ ⇒ while around signal for barging check
non blocking	$C < W < S$ (signal)	$C = W < S$ (Java / C#) ⇑ while around signalled wait() for barging check
implicit signal	$C < W$	$C = W$

good for prototypes, but bad perf.

μC++ has _Cormonitor: coroutine <> monitor

Java Monitors

Use `synchronized` word to make class a monitor

```
class Foo {
    public synchronized <type> (name) () { ... }
}
```

Limitations:

① 2 fncts: wait(), notifyAll() → could lead to barging

② 1 condition queue → makes certain problems impossible

③ Spurious wakeup: waiting threads can randomly take up & so check if allowed → need to have while loop around wait()

Sdn: ticketing for barge control & while loops

Also suffers from nested monitor problem

CONCURRENT ERRORS

Race condition → poor M.E.
→ poor synchronization
} threads run assuming in correct state.

NO progress

① Livelock

Indefinite postponement on simultaneous arrival

Can always break livelock w/ some tie break

② Starvation

A task never executes b/c another stream of tasks always beats

Long-term starvation is rare

③ Deadlock:

Processes waiting for event that will never occur. Synchronization / M.E.

5 conditions:

A: concrete resource requires M.E.

B: process holds resource & waits for access held by someone else

C: No pre-emption of resource (cannot take away resource)

D: Circular wait

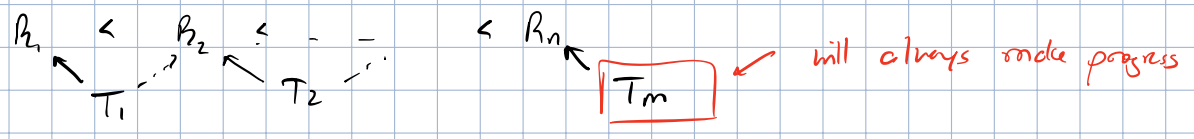
E: simultaneous

Deadlock Prevention

Synchronization deadlock → impossible to statically prevent

Mutual exclusion deadlock:

1. No M.E. → impossible
2. No hold & wait: only give all resources → poor resource utilization
3. Allow pre-emption of resources → cannot do this statically
4. No circular wait: ordered resource policy
   - ① Divide resources into classes
   - ② Only request resource from class R_i if not holding resource for any class R_j $(j \geq i)$



5. Prevent simultaneous occurrence

Deadlock Avoidance

Obj: dynamically detect if deadlock occurs → prevent situation

Banker's Algo

Check's if safe sequence of allocs → no deadlock. Run algo on each alloc.

① Define total amount of resources in sys.

R_1	R_2	R_3	R_4	} TC
6	12	4	2	

② Define max. # of resources needed / thread.

	R_1	R_2	R_3	R_4	} M
T_1	4	10	1	1	
T_2	2	4	1	1	
T_3	5	9	0	1	

③ Define current allocated resources.

	R_1	R_2	R_3	R_4	} C
T_1	2	5	1	0	
T_2	1	2	1	0	
T_3	1	2	0	0	

④ On resource request, update C w/ request. Calculate available resources (TC - C)

$C_{1,1}$: 2 → 3

	R_1	R_2	R_3	R_4	} CR
CR - C =	1	3	2	2	

⑤ For each thread → figure out how many more resources needed to max (N = M - C)

	R_1	R_2	R_3	R_4	} N
T_1	1	5	0	1	
T_2	1	2	0	2	
T_3	4	7	0	1	

⑥ Choose thread order s.t. all CR resources can be allocated & fulfill some thread's max value (i.e. $CR - N_i = 0$)

Choose $T_2 \Rightarrow$ alloc all resources would make $N_2 = 0$ ($CR - N_2 \geq 0$)

⑦ Give back that thread's resource to CR

CR:	R_1	R_2	R_3	R_4
	2	5	3	2

Repeat until CR = TC (all threads could be run to max & give back)

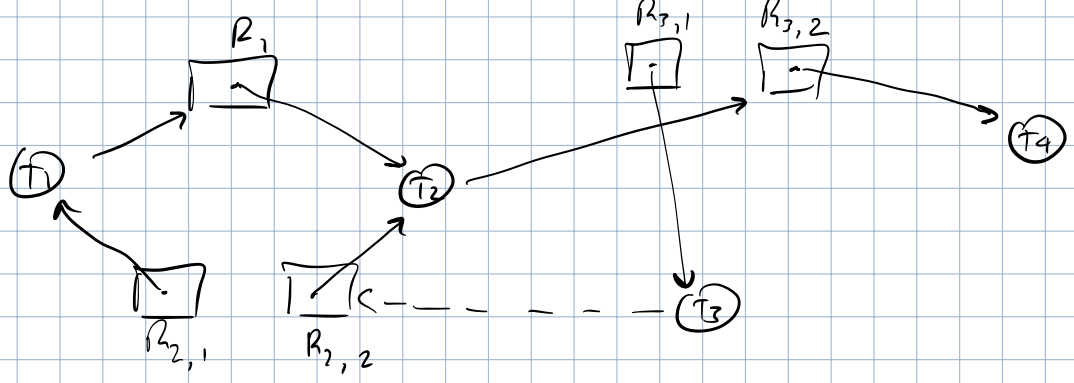
Allocation graphs

Legend:

Create graph of allocation. No cyc ⇒ no deadlock. If multiple instances of resource, then cycle ⇏ deadlock

↳ Check isomorphic graphs when 1 node = 1 thread / 1 resource instance then cycle ⇒ deadlock

Ex:



Check resource allocation graph on alloc attempt & simulate a path to no deadlock:

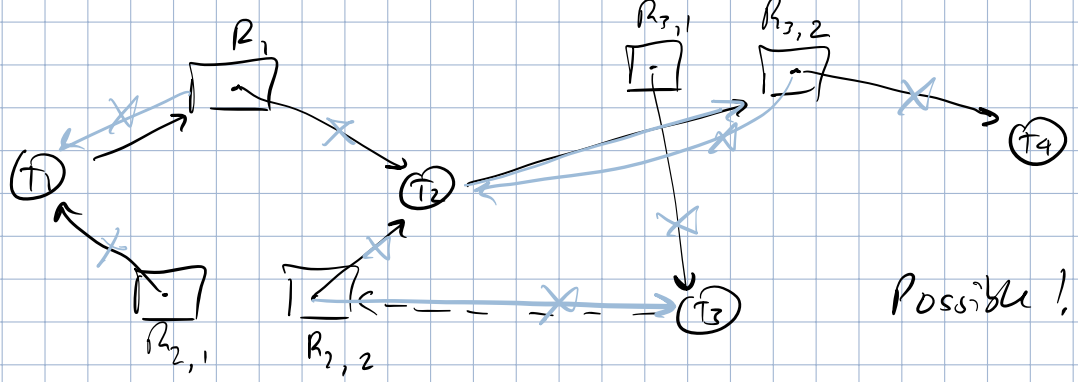
① Check isomorphic

② Thread that is not in hold & wait (eg. T_4). Release resource.

③ Give resource to T_i

④ Repeat → remove all arrows. If possible → no deadlock

Ex:

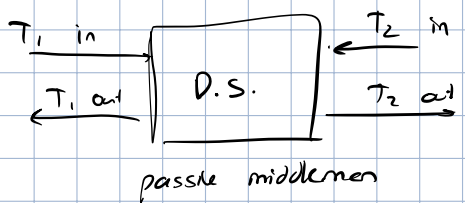


Deadlock Detection & Recovery

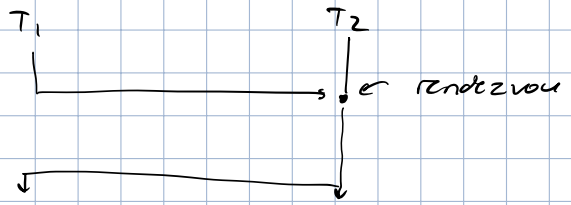
Let deadlock happen → recover. Need to build allocation graphs & check cycle every T sec. Need preemption

DIRECT COMMUNICATION

Objective:



⇒



* Key: a _Task is like a _Monitor → only 1 thread can be active in _Task @ a time
  - Public members already have _Mutex
  - If T_1 calls _Task public member → _Task can block & allow caller to execute via external sched

How to make a Task into monitor:

```
_Task A {
    private:
        ...
    public:
        void foo() {...}
        void bar() {...}
    private:
        void main() {
            for(;;) {
                _When (<cond>) _Accept (foo) {...}
                or _When (<cond>) _Accept (bar) {...}
                :
            }
    }
```

Should not need to _Accept. Make this as small as possible → _Task thread does work

_Task Thread

gets blocked → other threads can call public method

ran by _Task after call done

Conditional _Accept

}

Notes:

① If multiple calls accepted & outstanding calls to call (eg. _Accept(foo & bar))
↳ Execute {..} of earliest _Accept → order matters

② {..} execute in 2 situations:
A: function accepting finishes
B: caller calls wait() in function that we _Accept

③ If using internal scheduling, _Task thread must call signalBlock()

④ To stop monitor, do: _Accept(~className) { return; }
↳ Caller gets blocked while {..} executes → ~className() postpone

Increasing Concurrency

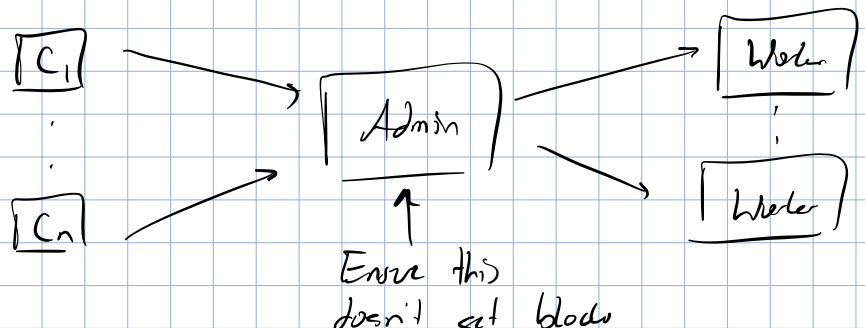
Client ⟷ Server } increase concurrency on both

Q: How to increase concurrency on server-side?

Default _Task method has a problems:

① M.E. of function ⇒ client will get blocked waiting for server to do work

Idea: separate out admin from work



Type:
- Timer
- Notifier
- Simple worker
  ↳ Takes work & returns result
- Complex worker:
  ↳ returns to client
- Courier: does blocking calls

Notes:

① Admin can have bench for workers to wait while work is getting ready

② Admin will never have deadlock

③ Client will drop off work to Admin in internal buffer

Q: How to increase concurrency on client-side?

Obj: asynchronous calls to server → client shouldn't wait.

↳ In μ++: no async calls → build async. out of sync. calls

Methods:

① Value returns

Separate out call & returns

```
server.call();   } can be done asyn. to server
work             }
server.wait();   ← could block
```

Issue: server needs to know which client is waiting to give proper result

② Tickets

Do same thing but server gives client ticket to identify client

Issues: assuming no malice from client (eg. ticket is failure pair)

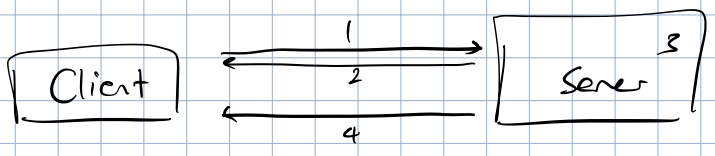
③ Call-back routines

Transmit routine on call → server will call routine on work completion

Assumes call-back is not blocking

Client must wait/poll until call-back called, "push" result from server

④ Futures



1 : client asks for result
2 : server immediately gives future
3 : server computes result
4 : server delivers result to future in client

If client wants to use future, must wait for step 4.

General design:

Future wraps result

```
class Future : public ResultType {
    friend _Task server;        → server can change future (step 4)
    ResultType result;
    uSemephore avail;
    Future * link;              → linked list of futures, for server to fill in
  public:
    ctor. . .
    ResultType get () {
        avail.P();              → Caller blocks here. Server must signal that
        return result;            result is ready
    }
}
```

μ++ futures:

Client:

```
#include <uFuture.h>
.
.
Future_ISM<ResultType> foo;
foo = server.perform(. . .);
.
.
ResultType result = foo();      ⇒ wait on future. This must be done!
.
.
foo.reset();                    → make future empty & restart
.
.
foo.cancel();                   → Stops server & client from using it
.
.
foo.available();                → T if async call complete, F o.w.
```

Declaration & get future

Pointers of futures cannot change

Server

① Keep reference of created futures

② Deliver result to future

```
int work = ...
future.deliver(work);
```

Assuming work matches ResultType of future

Can also deliver exception

```
future.deliver(new Exception());
```

Future will handle dynamic frees for you

Complex Access

_Select (f1) { .. } or _Select (f2) { .. } and _Select (f3) { .. }

{..} executed on f1 delivery

Both f2 & f3 must be delivered to move forward BUT {..} is executed on single delivery

If select sequence not satisfied → block

Conditional select:

_When (..) _Select (..)
or _When (..) _Select (..)
and _When (..) _Select (..)

} toggle _Select on & off

_Else also available → runs if _Select statement block.

OPTIMIZATIONS

Types of optimizations

- Reordering: Data & code reordered for perf.
- Eliding: Removing unnecessary code & data
- Replication: Duplicate across resources (eg. concurrency)

Goal: faster prog. but should be isomorphic (input → output is same)

Note: more optimizations → compile time ↑

Sequential Optimizations

① Reorder disjoint operations

{ t = x; s = y; , y = 0; x = 3; ↓

② Elide output.

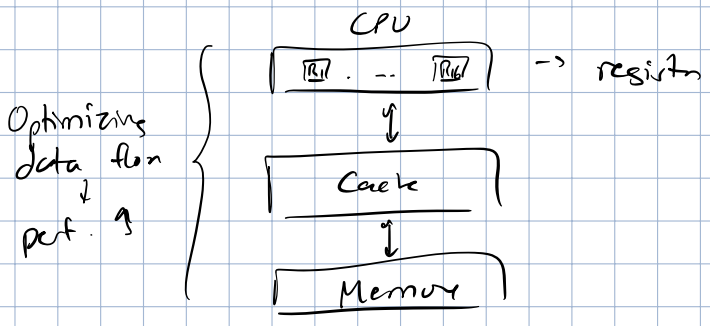
```
x = 0;
x = 3;
```

~~`for (int i=0; i<n; i++);`~~

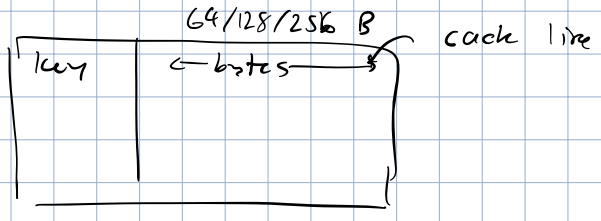
③ Execute code in parallel if multiple units (eg. get int add & float add in diff adders)

↳ Micro-parallelism in H.W., limits in sequential

Memory Hierarchy

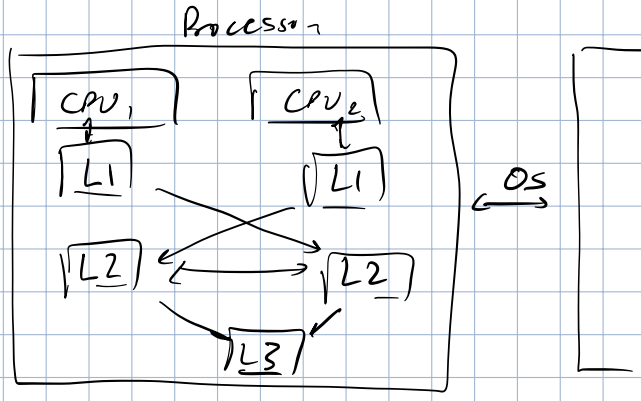


Cache: fast access to mem. that cannot be stored in registers for time



- Reordering: enables faster access to data based off cache

Cache coherence:



Q: How to reconcile cache diff. across cores / processors?

A: Cache coherence protocol from H.W.

Cache consistency: CPUs see data in an order that makes sense

Consistency + coherence ⇒ atomic R/W

↓

Take time ⇒ threads will read stale values. Block until values updated

Multithreading problems w/ caches:

① Cache thrashing: multiple threads update same cache line → cache update ↑

② False sharing: multiple variables thrash as entire cache line thrashes

Concurrency Optimizations

Sequential prog. have strong memory ordering: value reading is always last value written

Concurrent " weak " : not guaranteed to be reading right value

Concurrent prog.:

concurrent code
Seq. code } optimize, but can still cause errors
conc. code

Examples of seq. optimizations → concurrent errors:

① Disjoint ordering

A: Reordering reads → not a problem

B: Reordering reads b/f writes → problem

C: Reorder write b/f read → problem

D: Reordering writes → problems

E: Reorder locks acquisition & release → problems

↳ Double-checked locking: only allow 1 thread to set ip

```
int* ip = nullptr;
...
if (ip == nullptr) {
    lock.acquire();
    if (ip == nullptr) {
        ip = new int(...);
    }
    lock.release();
}
...
```

Check twice?
2 threads come in, both see ip == null, both will try to change if 2nd not there

This doesn't work! Compiler reorders malloc & setting ip to malloc. so multiple threads can still set ip!

② Eliding

Remove code used in other threads → problems.

Memory Model

Manufacturers set model → which optimizations H.W will perform

↳ Compiler can use this to prevent optimizations from messing concurrent prog.

Minimum model for concurrency: sequential consistency

Prevent Optimization Problems

Goal: race-free prog. (produce races if M.E & sync provided)

↳ Doesn't mean no races. Races intrnl to crit. section (up to prog.)

Approaches:

① Augment code w/ pragmas → easier

② Formally define races & prevent

Tricks:

① volatile: forces load & store of variable

↳ In Java / C++: atomic

② Set fences: bounds compiler reordering

Hard!

OTHER APPROACHES

Atomic / Lock-Free Data Structures

Obj: no ownership over D.S. but still only allow atomic changes

Misnomer ⇒ still spinning. If eventual progress → wait free D.S.

Based on atomic CAS (compare & set):

```
bool CAS (int& val, int comp, int nval) {
    if (val == comp) {
        val = nval;
        rtrn true;
    }
    rtrn fals
```

If val == comp, set.
Returns if val == comp

}

Ex: // Stack:

```
void Stack::push(Node& n){
    for(;;){
        n.next = top;
        if(CAS(top, n.next, &n)) break;
    }

Node* Stack::pop(){
    Node* t;
    for(;;){
        t = top;
        if(t == nullptr) return t;
        if(CAS(top, t, t->next)){ break;
    }
}
```

Ensures top = n atomically

Problem: ABA

Stack: top → A, B, C]

① T_1 calls pop, t = A. B/f CAS, time sliced

② T_2 pops A, B, pushes A

Stack: top → A, C]

③ T_1 returns. Check if top == A → true! Set top to B (but B already popped!)

Fix:

① New atomic instr.

double-wide CAVD atomic instr.

```
bool CAVD(int& val, int& comp, int nval){
    if(val == comp){
        val = nval;
        return true;
    }
    comp = val;
    return false
}
```

② Have tickets for each node on stack

Every push has incremented count on data struct. Only pop if right counter.

Only probabilistic situation

Exotic atomic instructions

H.W. provide stronger atomicity → create more complex lock-free D.S.

Ex:// MIPS has LL to put value into reg., SC to write back
↳ set a reservation → tells if reg. overwritten!

On SC → error if oracle says reg. is overwritten.

GPU

GPU uses SIMD arch ⇒ all instr. on CPUs must occur at same time

Makes prog. difficult, esp. branching ⇒ must execute all instr. even if branch = false!
↳ Requires complex changes to prevent unless ops + speedups

No cache, only registers.

Great for vectorized ops.

Other Languages

Ada

① When needs to be @ top & can only use class variables

② No internal scheduling (all wkrs, implicit signalling)

③ Problem w/ blocking (requeue after computation → lose everything!

Java

① Requires start() & join() to be called

Go

① Non-pre-emptive user threads

② Channel comm. vs. object comms. in μ++

C++

① Call:
thread objName (routine, params);

② Need to call join()

③ How many concurrency constructs but monitor is buggy

java.util.concurrent

Multiple condition locks!

pthreads