CSU23016

Concurrent Systems
&
Operating Systems

Andrew Butterfield
ORI.G39, Andrew.Butterfield@scss.tcd.ie

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

with thanks to Mike Brady

Student Online Teaching Advice Notice

The materials and content presented within this session are intended solely for use in a context of teaching and learning at Trinity.

Any session recorded for subsequent review is made available solely for the purpose of enhancing student learning.

Students should not edit or modify the recording in any way, nor disseminate it for use outside of a context of teaching and learning at Trinity.

Please be mindful of your physical environment and conscious of what may be captured by the device camera and microphone during videoconferencing calls.

Recorded materials will be handled in compliance with Trinity's statutory duties under the Universities Act, 1997 and in accordance with the University's policies and procedures.

Further information on data protection and best practice when using videoconferencing software is available at https://www.tcd.ie/info_compliance/data-protection/

Overview

- Concurrency
  - What / Why / How,
  - Grappling with Concurrency Issues.
- Operating Systems
  - Operating system architectures,
  - Memory Management (OS perspective),
  - Processes and Thread Management,
  - File Storage (disk I/O and file systems).

Practical Matters — Assessment

- 20% Assignments
  - Tool Usage 2%
  - Thread Programming 6%
  - Concurrency Modelling 6%
  - OS Coding 6% (Scheduler, Filesystem, ...) 6%
- 80% Examination
  - 2hour exam in a 24 hour window

Practical Matters — Linux

- Course is based on Linux and the standard C program build toolset, "Build Essentials", POSIX Threads, Mac OS X also works pretty well.
- If you use Linux all will be well with all assignments
- If you don't use Linux (most of you?), then Assignment 1/2 are problematic
  - Suggest you either:
    - Develop your work on MacNeill or one of the LG12 machines
    - consider running Linux in a Virtual Machine - Virtual Box, say.
- Assignment 1 is simple to help smoke out issues
- Assignment 2 is tricky

POSIX

- POSIX – Portable Operating System Interface
  - for variants of Unix, including Linux
  - IEEE 1003, ISO/IEC9945
- Really, considered a standard set of facilities and APIs for Unix.
  - 1988 onwards
  - Doesn't *have* to be Unix
    - macOS is certified to support POSIX, but ...
    - Windows: partial POSIX compliance with CygWin, Window Subsystem for Linux (WSL)
  - ref: http://en.wikipedia.org/wiki/POSIX

POSIX Threads

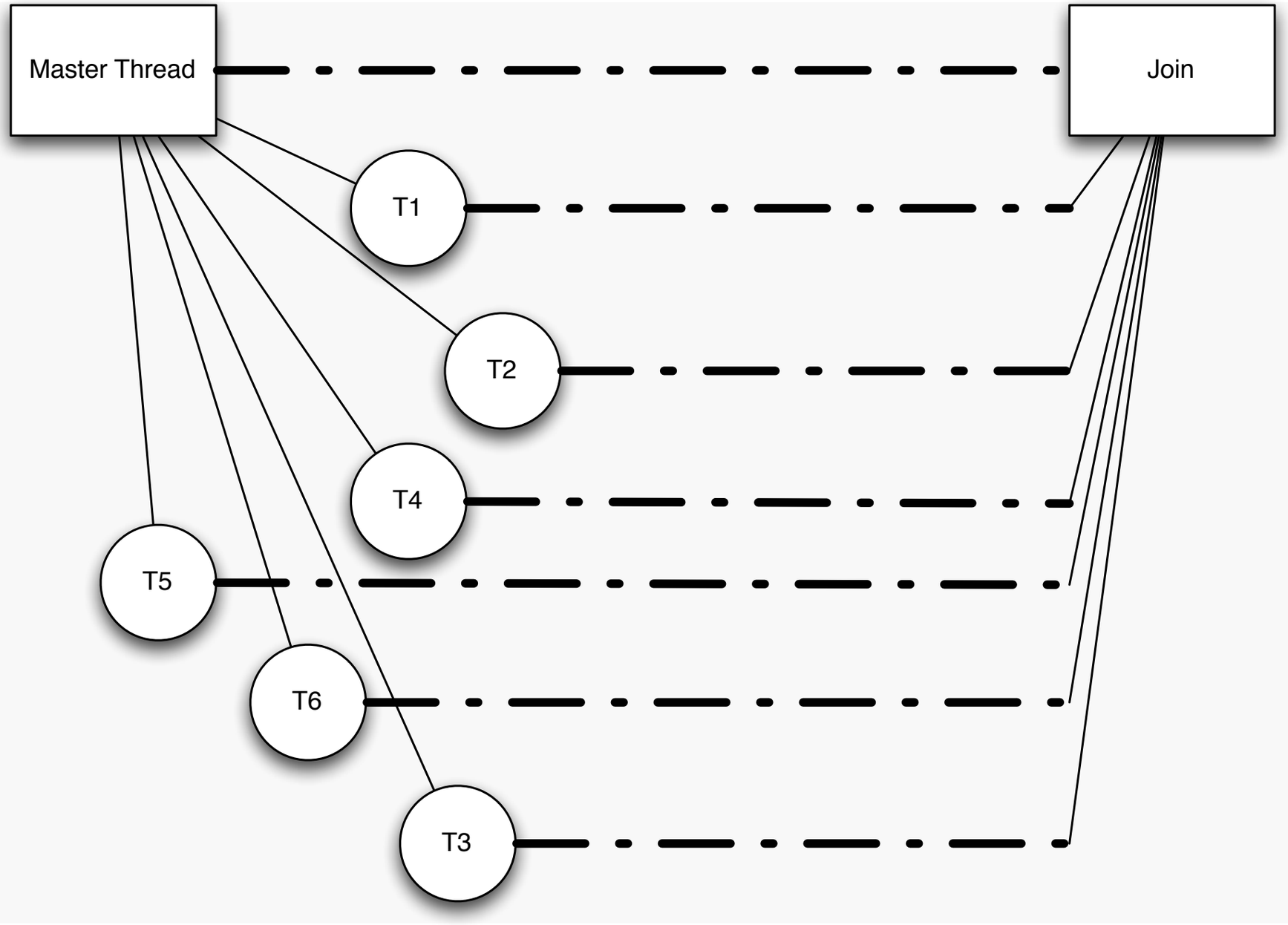
- POSIX Threads aka *'pthreads'*
  - correspond to 'Light Weight Processes' (LWPs) in older literature.
  - *pthreads* live within processes.
    - *processes* have separate memory spaces from one another
      - thus, inter-process communication & scheduling may be expensive
    - *pthreads* (within a process) share the same memory space
      - inter-thread communication & scheduling can be cheap

Classic tradeoff: speed vs. stability/ruggedness

POSIX Threads (2)

- Portable Threading Library across Unix OSes
  - All POSIX-compliant Unixes implement *pthreads*
    - Linux, Mac OS X, Solaris, FreeBSD, OpenBSD, etc.
- Also Windows:
  - E.g. Open Source: *pthreads-win32*
  - But these are different, so are not suitable for assignments
- OS X is very strict
  - OS X implementation if pthreads is very strict - it can fail ways that Unix pthreads won't.

Six Separate Threads



Creating a pthread

```
#include <pthread.h>

int pthread_create(
      pthread_t *thread,            // the returned thread i
      const pthread_attr_t *attr, // starting attributes
      void *(*start_routine)(void*),
                        // the function to run in the thread
      void *arg         // parameter for function
);
```

Where…

- '*thread*' is the ID of the thread
- '*attr*' is the input-only attributes (NULL for standard attributes)
- '*start_routine*' (can be any name) is the function that runs when the thread is started, and which must have the signature:

```
void* *start_routine (void* arg);
```

- '*arg*' is the parameter that is sent to the start routine.
- returns a status code. '*0*' is good, '*-1*' is bad.

Wait for a thread to finish

- `int pthread_join( pthread_t thread,`
  `void **value_ptr`
  `);`

- where
  - *'thread'* is the id of the thread you wish to wait on
  - *'value_ptr'* is where the thread's exit status will be placed on exit (`NULL` if you're not interested.)
  - NB: a thread can be joined only to one other thread!

Thread Code

```
void *PrintHello(void *threadid) {
     printf("\n%d: Hello World!\n", threadid);
     pthread_exit(NULL);
}
```

- `threadid` is a pointer to a number (printed using %d)
- `pthread_exit` tells pthreads that has finished, and returns a pointer to a return value, if required.

Hello World -- Creating Threads

```
int main (int argc, const char * argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc,t;
    for (t=0;t<NUM_THREADS;t++) {
        printf("Creating thread %d\n",t);
        rc = pthread_create(&threads[t],NULL,PrintHello,(void *)t);
        if (rc) {
            printf("ERROR return code from pthread_create(): %d\n",rc);
            exit(-1);
        }
    }
    ...
```

Hello World -- Waiting for Exit

```
...
// wait for threads to exit
    for(t=0;t<NUM_THREADS;t++) {
        pthread_join( threads[t], NULL);
    }
    return 0;
}
```

HelloWorld -- Complete

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS 6

void *PrintHello(void *threadid) {
    printf("\n%d: Hello World!\n", threadid);
    pthread_exit(NULL);
}

int main (int argc, const char * argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc,t;
    for (t=0;t<NUM_THREADS;t++) {
        printf("Creating thread %d\n",t);
        rc = pthread_create(&threads[t],NULL,PrintHello,(void *)t);
        if (rc) {
            printf("ERROR return code from pthread_create(): %d\n",rc);
            exit(-1);
        }
    }
    ...
    // wait for threads to exit
    for(t=0;t<NUM_THREADS;t++) {
        pthread_join( threads[t], NULL);
    }
    return 0;
}
```

MacNeill/LG12/Ubuntu

- Compile hello.c as pthreaded executable "hello":

```
cc -o hello hello.c -pthread
```

  - Include the pthread library to allow it to compile and link
  - Sometimes, `-lpthread` works, but `-pthread` is correct
  - pthread library automatically included in Mac OS X build

- To run program "hello":

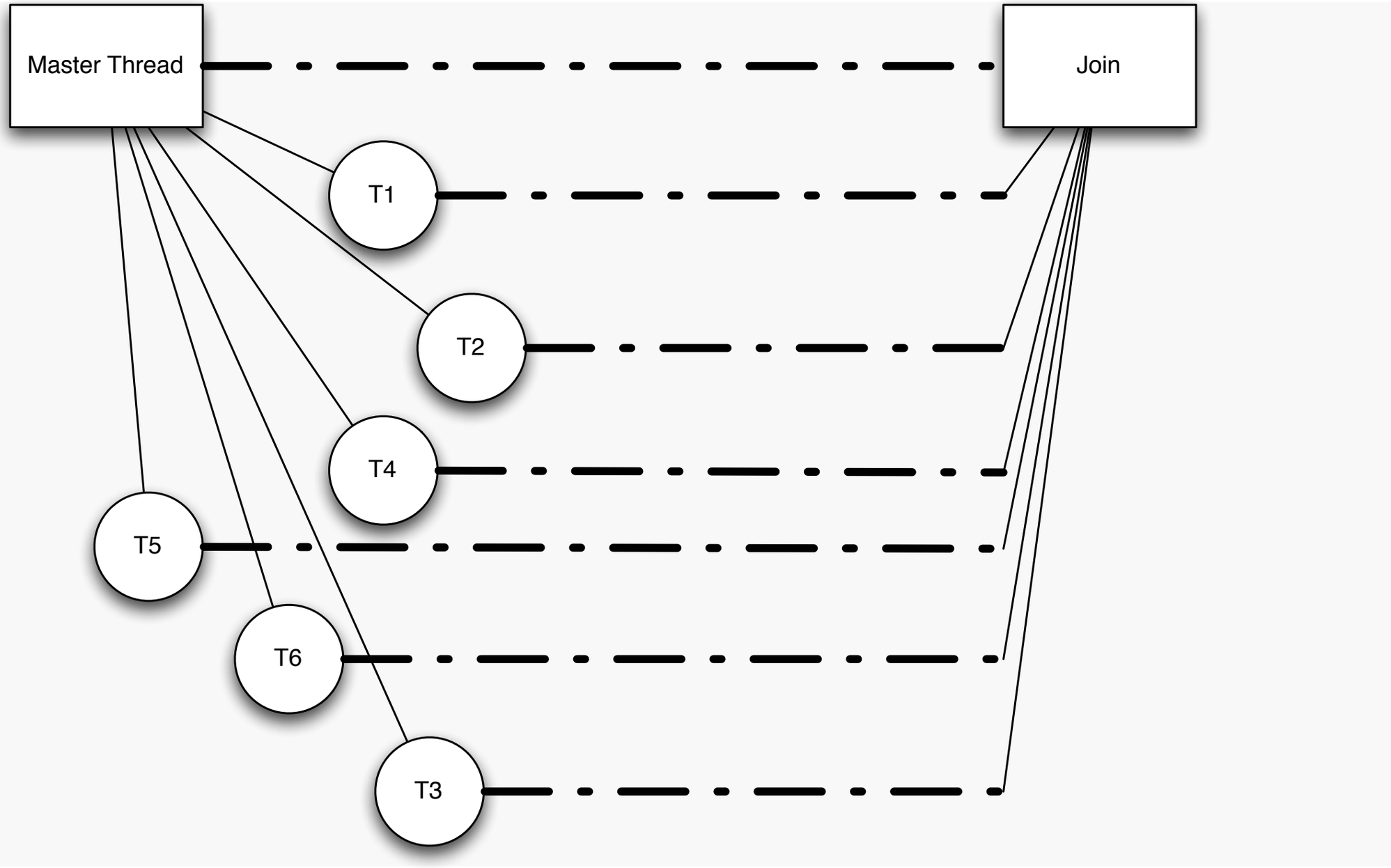
```
./hello
```

  - Let's try it (DEMO)

HelloWorld -- Running it

- The order in which each thread is created is always the same
  - Determined by the for-loop we use to call `pthread_create`
- The order in which each printf statement executes varies
  - A lot!
  - Because ? .....

What's curious about this?



What if ... ?

- The threads interacted in some way?
  - e.g. each thread increments a global variable that tracks the number of threads that ran?
  - What could possibly go wrong?
- Consider a variant of hello.c that does this (`sumofhellos.c`)

sumofhellos.c

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

#define NUM_THREADS 6
#define USERNAME "username"

int x; // Global variable !!

void *PrintHello(void *threadid){
    int y ; // (Thread) local variable

    y = x; // read that global
    printf("\n%d: Hello World, from %s!\n",threadid,USERNAME );
    x = y+1; // write that global
    pthread_exit(NULL);
}

int main(int argc, const char * argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc,t;
    x = 0; // Initialise that global !
    for (t=0;t<NUM_THREADS;t++){
        printf("Creating thread %d\n",t);
        rc = pthread_create(&threads[t],NULL,PrintHello,(void *)t);
        if (rc) {
            printf("ERROR return code from pthread_create(): %d\n",rc );
            exit(-1);
        }
    }
    // wait for threads to exit
    for(t=0;t<NUM_THREADS;t++){
        pthread_join(threads[t],NULL);
    }
    // Display that global !
    printf("\nAll threads done by %s, x = %d\n",USERNAME,x);
    exit(0);
}
```

What should be
the final value of x ?

DEMO !

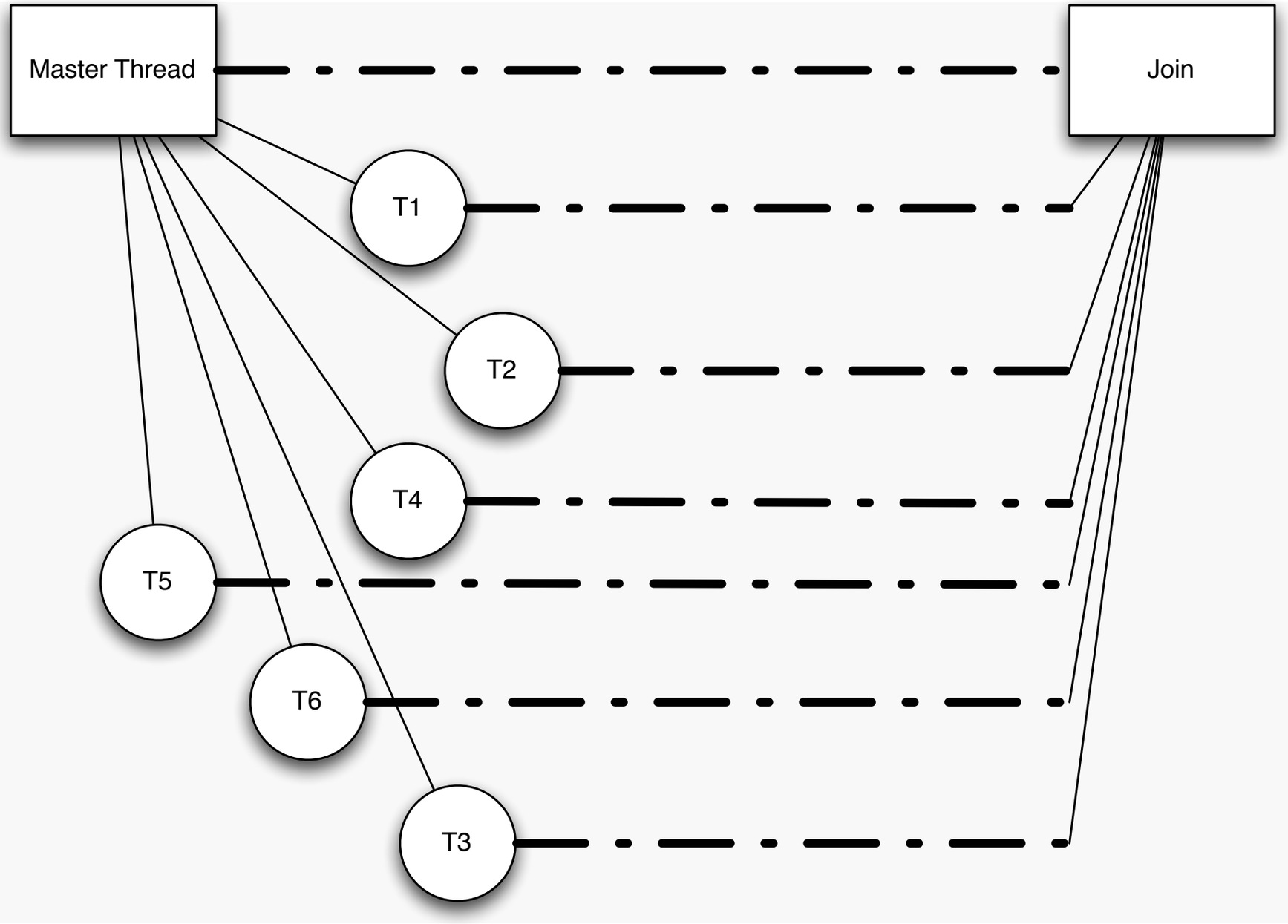
Woah! Hold on

- The final value of x should have been 6.
  - The final value of x after a run varied, most often 3, 4, or 5
  - Less often, 1,2, or 6.
- What is going on?

Runtime Behaviour

- The Runtime Behaviour of the Program is no longer under the control of the program.
  - The order in which work gets done on the machine is not exactly under the control of the program
  - What we observe is an (apparently) arbitrary interleaving of (atomic) actions by each running thread
  - It seems to be a price that's paid for parallelism, but:
    - What errors can it introduce?
    - Can we prevent them / protect against them / design them out?

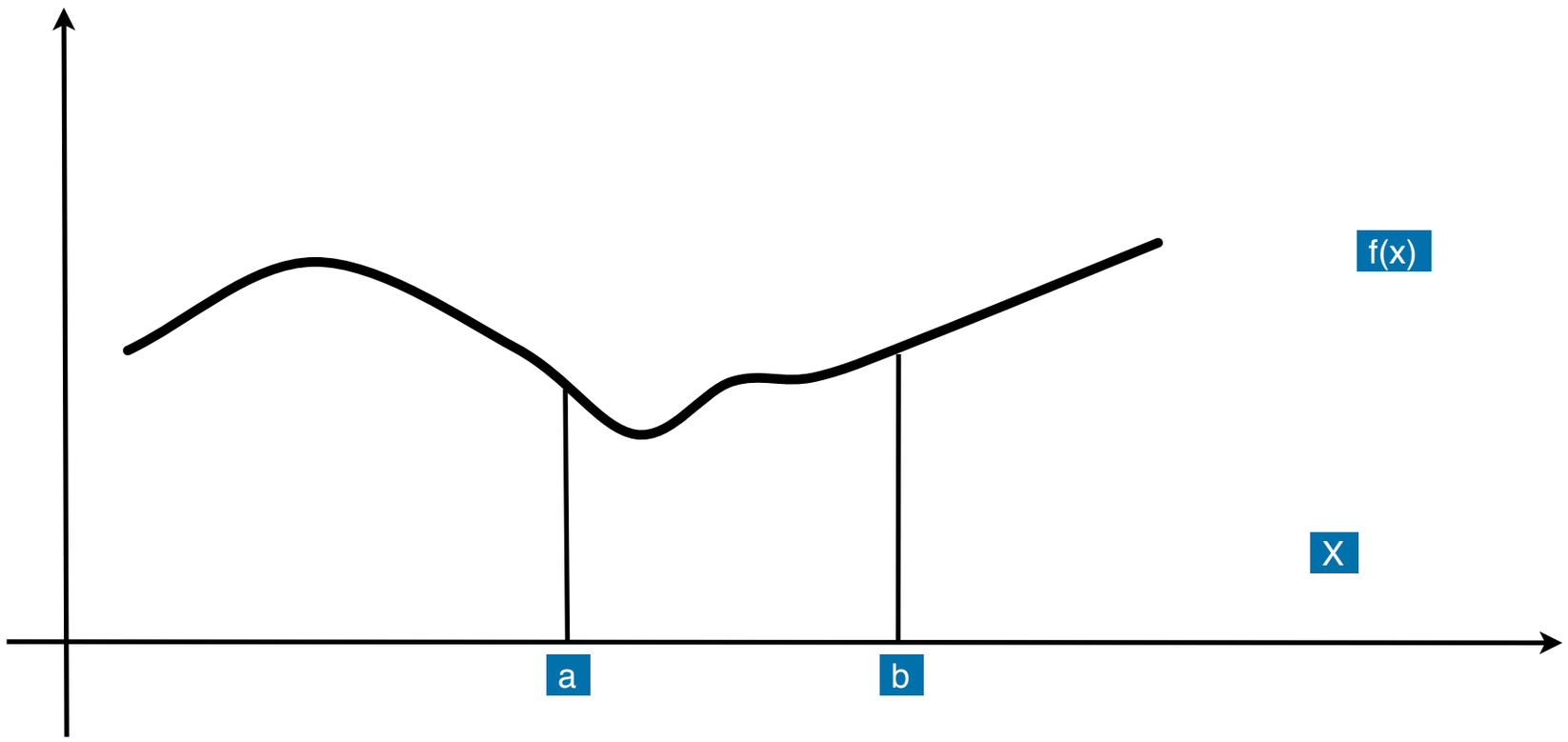
What's ~~wrong~~ deceptive about this?



“Massively” Parallel Problems

- A problem is said to be "Massively" Parallel if:
  - there are almost no interaction between parallel threads calculating independent parts of the solution.
  - So, in theory, everything can be run at the same time.

Example: Numerical Integration



Numerically integrate from a to b

$$\int_0^4 (16 - x^2)dx = 42.6666\ldots$$

- Consider integrating f(x) = 16 - x^2 between 0 and 4
- The analytic solution is 128/3 or 42.666…(optional exercise: check my math)
- We break the interval [0..4] into a thousand vertical slices (`H`=0.004),
  - use the "trapezoidal rule" to calculate the `area` of each slice
  - sum all of these `areas` into an `answer` variable.
- Running code in `seq-integrate.c` gives the answer: 42.666656

integration preamble

```
#include <stdio.h>

#define NUM_SLICES 1000
#define H 4.0/NUM_SLICES

double answer;

double f(double x) { return (16.0 - x*x) ; }

double trapezoid(double a, double b) { return H*(f(a)+f(b))/2.0; }
```

seq-integrate.c body

```
int main (int argc, const char * argv[]) {
  answer    = 0.0;
  double a ;
  double b ;
  int i ;
  for (i=0;i<NUM_SLICES;i++) {
    a = (int)i * H ;
    b = a + H;
    answer += trapezoid(a,b);
   }
  printf("\nAnswer is %f\n",answer);
    return 0;
}
```

Integration with threads (attempt 1)

- We might use 1000 threads, one for each of the trapezoid calculations.

nom-integrate.c thread

```
void *IntegratePart(void *i) {

  double a,b,area;

  a = (int)i * H ;
  b = a + H;
  area = trapezoid(a,b);

  // critical section with no mutex !!!!!
  answer=answer+area;

    pthread_exit(NULL);
}
```

nom-integrate body

```
int main (int argc, const char * argv[]) {
    pthread_t threads[NUM_SLICES];
    long rc,t;

  answer = 0.0;

    for (t=0;t<NUM_SLICES;t++) {
        rc = pthread_create(&threads[t],NULL,IntegratePart,(void *)t);
        if (rc) {
            printf("ERROR return code from pthread_create(): %ld\n",rc);
            exit(-1);
        }
    }
    for(t=0;t<NUM_SLICES;t++) {
        pthread_join( threads[t], NULL);
    }
  printf("%f\n",answer);
    return 0;
}
```

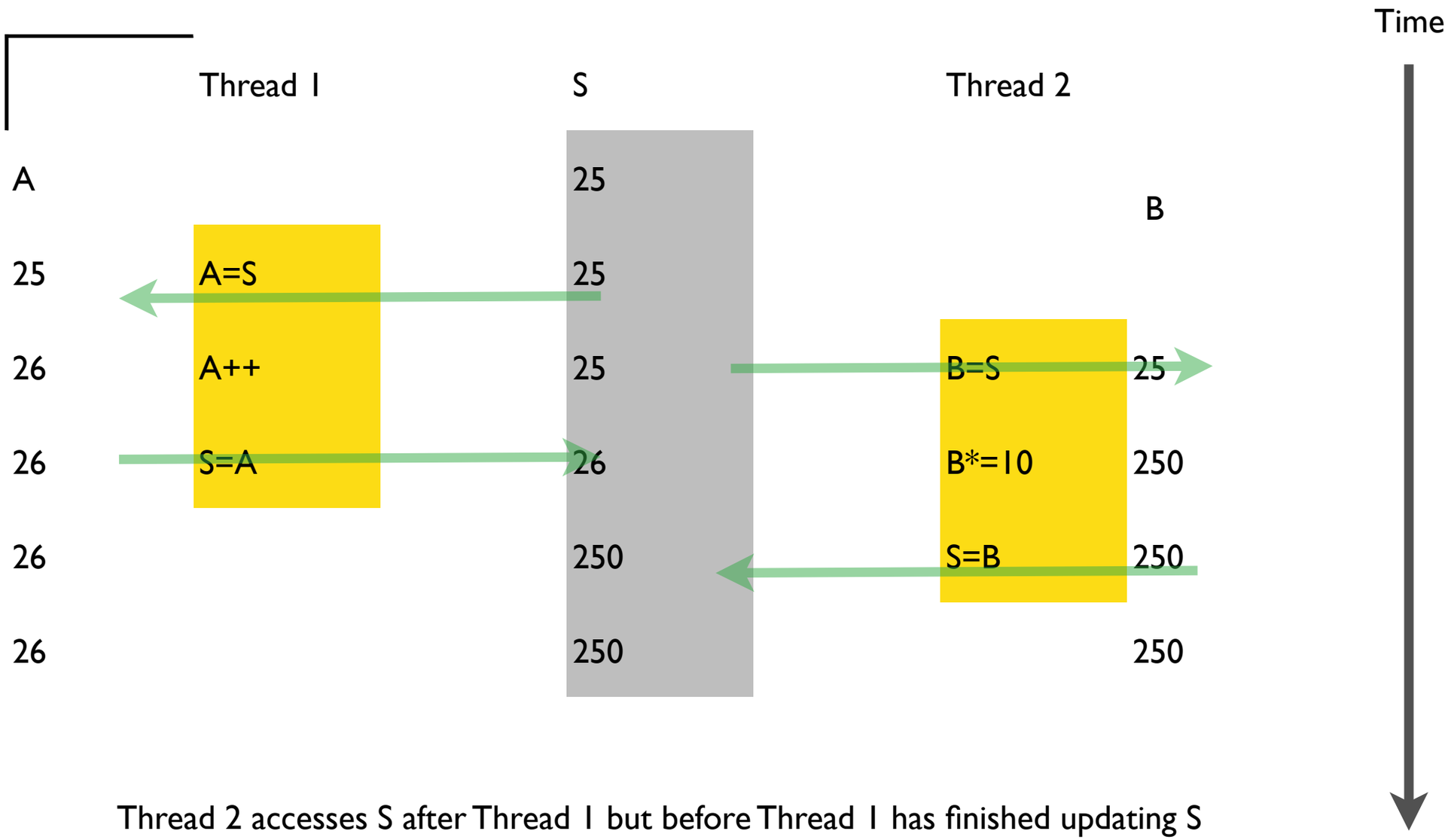
Integration with threads (attempt 1)

- When I ran this I got 42.666656, most of the time
  - (e.g. 46 times out of 52)
- However sometimes I got an answer in the range 42.6026 .. 42.6038 !
  - (e.g. 6 times of of 52)
  - Approximately 11% of the time I got an answer that underestimates by -0.15%
    - on my office iMac, at least.
- Hard to explain this result. May differ on different machines.
- Let's give it a go! (DEMO)

Synchronisation

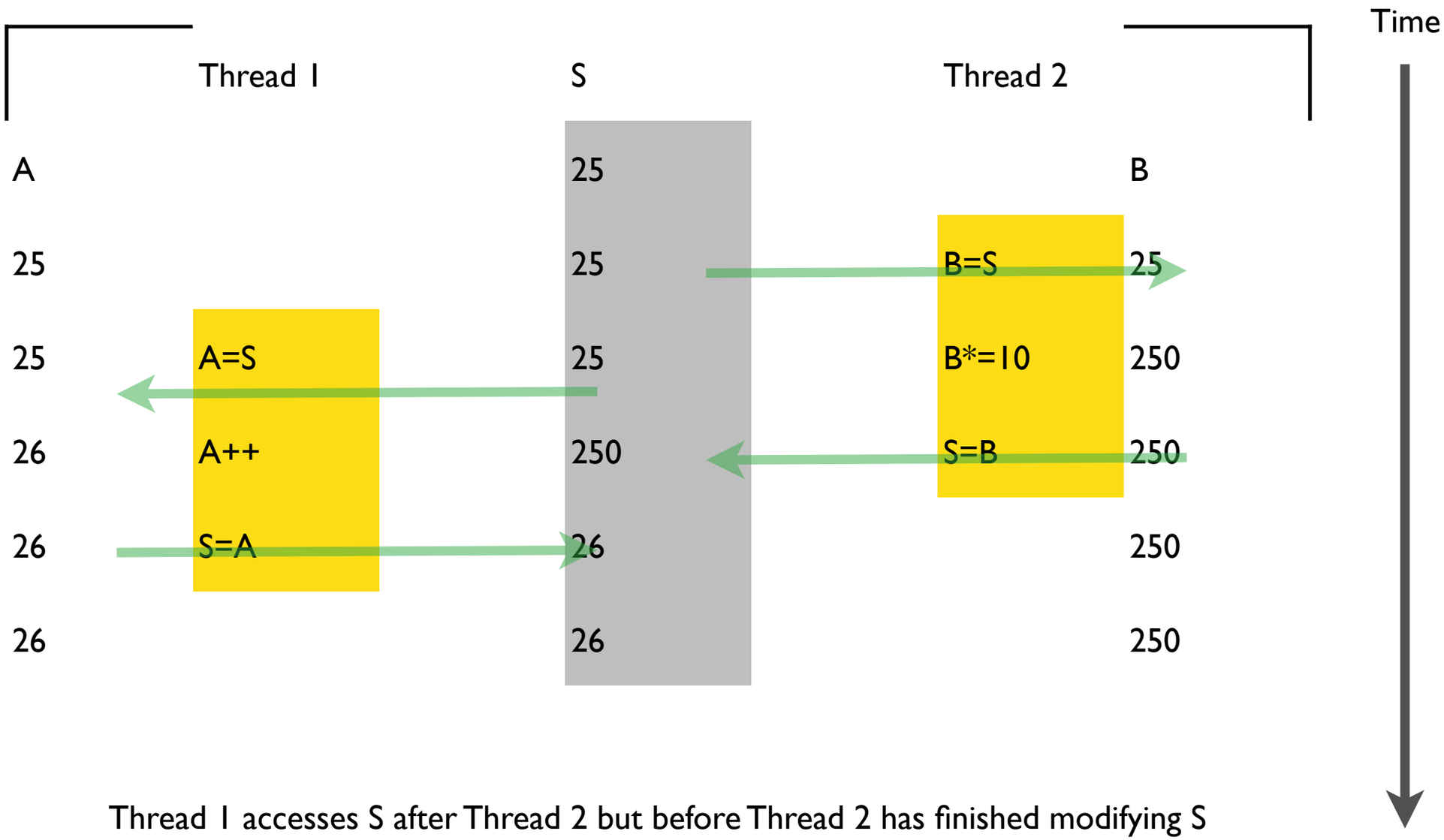
- With every thread we can associate a ‘write-set’:
  - the set of memory locations to which that thread writes.
- We’ve looked at situations where the threads can operate independently -- the ‘write sets’ of the threads don’t intersect.
- Where the write sets intersect, we must ensure that independent thread writes do not damage the data.

Shared Variable S: Thread 1 *before* Thread 2



Thread 2 accesses S after Thread 1 but before Thread 1 has finished updating S

Shared Variable S: Thread 1 *after* Thread 2



Thread 1 accesses S after Thread 2 but before Thread 2 has finished modifying S

Synchronisation

- Problem: Access to shared resources can be dangerous.
  - These are so-called 'critical' accesses.
- Solution. Critical accesses should be made exclusive. Thus, all critical accesses to a resource are *mutually exclusive*.
- In the example, both threads should have asked for exclusive access before making their updates.
  - Depending on timing, one or the other would get exclusive access first. The other would have to wait to get any kind of access.

Snchronisation

- A wide variety of algorithms have been devised that ensure synchronisation of competing threads:
  - "Semaphores", "Monitors", …
- All are very tricky to get right
- A helpful way to get simpler guarantees of correct behaviour is to have hardware support.
  - An "atomic" test-and-set instruction
    (read a memory location, check its value, and update the value if needed)
  - An "atomic" operation is one that either performs ALL its steps (without interruption)
    or does NONE of them.
    It never does just SOME of them.

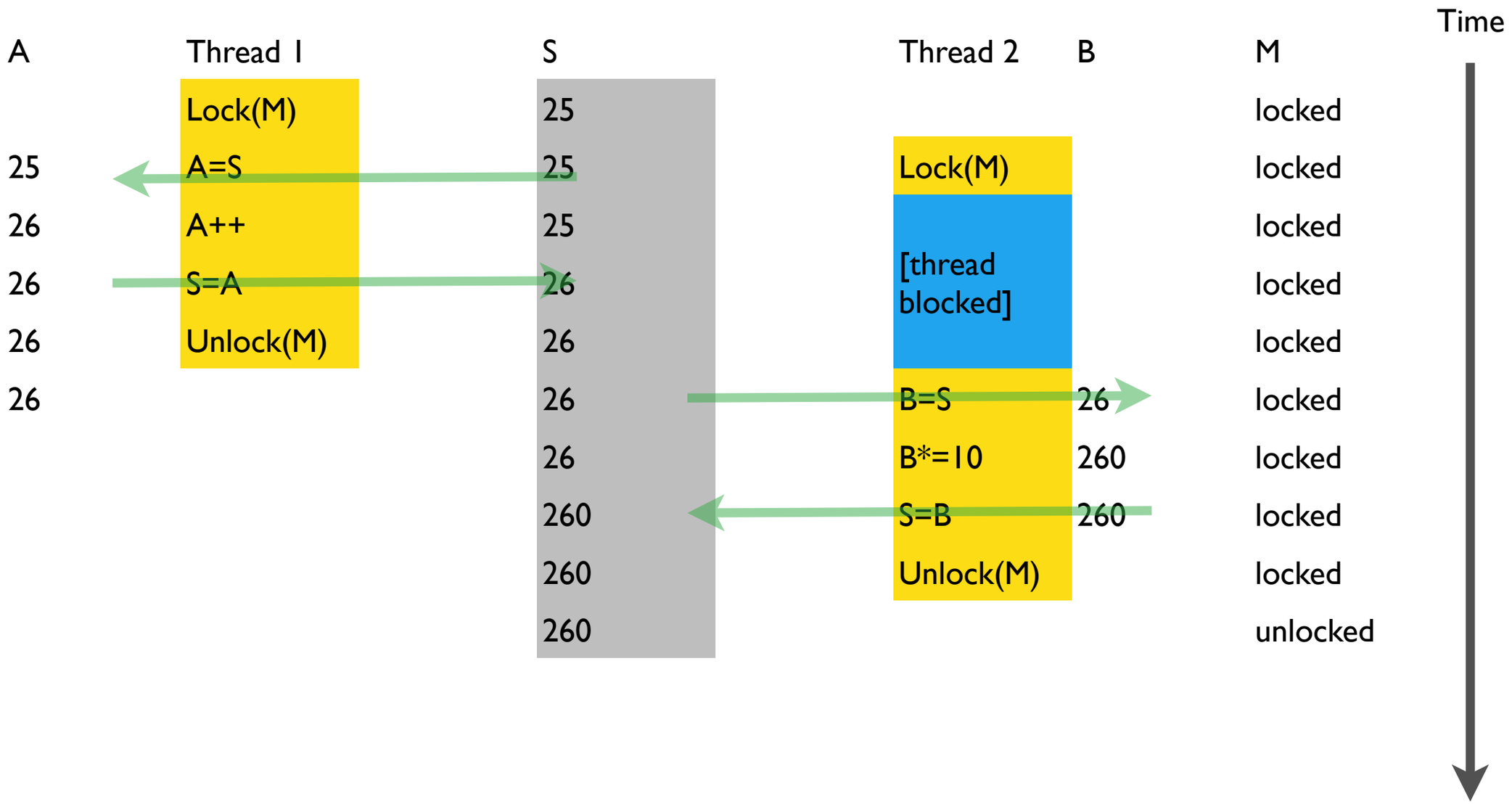
Ensuring Mutual Exclusion.

- Mutual Exclusion is a form of thread synchronisation
  - Used to manage access to shared resources (data, IO hardware, shared code, etc.)
- Key idea:
  - There is a "lock" associated with the resource
  - It is locked by a thread when it is using the resource, and unlocked by that thread when done
  - Any other thread must wait for it to be unlocked before proceeding to use that resource.
- accomplished using special "*mutex* variables".

Accessing a protected resource

- To access a mutex-protected resource, an agent must acquire a lock on the mutex variable.
  - If the mutex variable is unlocked, it is immediately locked and the agent has acquired it. When finished, the agent must unlock it.
  - If the mutex variable is already locked, the agent has failed to acquire the lock -- the protected resource is in exclusive use by someone else.
    - The agent is usually blocked until lock is acquired.
    - A non-blocking version of lock acquisition is available.
- A mutex variable is a complex data object
  - It needs to record if it is locked or unlocked (obviously).
  - It needs to record which thread has acquired the lock.
  - It also needs to keep track of which threads are waiting for it to be unlocked.

Shared Variable S Protected by Mutex M



Create pthread mutex variable

- Static:
  - pthread_mutex_t m = PTHREAD_MUTEX_INITIALISER;
    - Initially unlocked
- Dynamic
  - pthread_mutex_init(<ref to mutex variable>,attributes)

Lock and Unlock Mutex

- pthread_mutex_lock(<mutex variable reference>);
  - acquire lock or block while waiting
- pthread_mutex_trylock(<mutex variable reference>);
  - non-blocking; check returned code
- pthread_mutex_unlock(<mutex variable reference>);
  - Should only ever be called by the thread that has the lock!

Problems

- Voluntary
  - Mutexes 'protect' *code*, and only if all thread code uses them properly
  - The use of a mutex lock/unlock is a protocol that threads should follow so that code behaves well.
    - It cannot enforce good behaviour
    - Other programmers don't have to use them to get access to the protected resource
      - This is part of the tradeoff. Use processes rather than threads if you want better protection.
- Unfair
  - If multiple threads are blocked on a mutex, the order in which they waken up is not guaranteed to be any particular order.

$$\int_0^4 (16 - x^2)dx = 42.6666\ldots$$

- We break the interval [0..4] into a thousand vertical slices (`H`=0.004),
  - use the "trapezoidal rule" to calculate the `area` of each slice
  - sum all of these `areas` into an `answer` variable.
- Sequential solution in `seq-integrate.c` gives an answer of 42.666656
- Concurrent solution in `nom-integrate.c` usually gives the same answer
  - but occasionally gives results like 42.539102 or 42.602656 or 42.602853 or ….
- The problem arise when two threads try to perform

```
answer = answer + area ;
```

at almost exactly the same time.

Integration with mutex

- We finally decide to use a mutex (mutex-integrate.c)
- We get the same results as the sequential code.

mutex-integrate (mutex & thread)

```
pthread_mutex_t  mutex = PTHREAD_MUTEX_INITIALIZER;

void *IntegratePart(void *i) {

  double a,b,area;
  int rc;

  a = (int)i * H ;
  b = a + H;
  area = trapezoid(a,b);

  // critical section with mutex
  rc = pthread_mutex_lock(&mutex);
    checkResults("pthread_mutex_lock()\n", rc);

  answer=answer+area;

  rc = pthread_mutex_unlock(&mutex);
    checkResults("pthread_mutex_lock()\n",rc);

    pthread_exit(NULL);
}
```

Body is same as for `nom-integrate`

integration example wrap-up

- We can parallelise integration, but mutexes are required
- If we omit them, then errors may occur
- The worst case is when such errors are rare
  - such errors may not be revealed by testing
  - e.g. Mars Rover flash memory bug.
- What about speed?
  - Mutex usage statistics: http://0pointer.de/blog/projects/mutrace.html

CSU23016

Concurrent Systems
&
Operating Systems

Andrew Butterfield
ORI.G39, Andrew.Butterfield@scss.tcd.ie

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

with thanks to Mike Brady

Student Online Teaching Advice Notice

The materials and content presented within this session are intended solely for use in a context of teaching and learning at Trinity.

Any session recorded for subsequent review is made available solely for the purpose of enhancing student learning.

Students should not edit or modify the recording in any way, nor disseminate it for use outside of a context of teaching and learning at Trinity.

Please be mindful of your physical environment and conscious of what may be captured by the device camera and microphone during videoconferencing calls.

Recorded materials will be handled in compliance with Trinity's statutory duties under the Universities Act, 1997 and in accordance with the University's policies and procedures.

Further information on data protection and best practice when using videoconferencing software is available at https://www.tcd.ie/info_compliance/data-protection/

‘Shells’

- A ‘shell’ is a program that allows you to give commands to a computer and get responses.
  - Common GUI-based shells include ‘Finder’ for Mac OS X and ‘Explorer’ for Windows. These are easy to learn but hard to automate.
  - Command-line text-based shell programs are harder to learn, but support a high degree of automation. These are typically accessed through a "console window" program
    - called "Terminal" on Unix/Linux/macOS
    - several, such as "Cmd" or "Powershell", on Windows
  - Common UNIX shell languages are SH, BASH, CSH, TCSH, ASH, ZSH. On Windows, the provided shells use a custom language, and a totally different approach to running programs. However, UNIX shell support is possible
    - try using "Cygwin" (http://www.cygwin.com) or "Windows Subsystem for Linux" (WSL)
    - Or use PuTTY (https://www.putty.org), or similar to make an SSH connection to macneill.scss.tcd.ie

The bash shell

- A early shell, called "sh" was developed for an early version of Unix,
  - by Stephen Bourne, at AT&T.
- A modified free version, called "bash" was written for Linux
  - It's the "Bourne-Again Shell" !
- We will use this shell for this course.
- The Shell Command-Line/Scripting Primer
  - see links in READING content area on Blackboard

Important shell principles

- Shell commands generally invoke programs to so the work:
  - e.g. when you write `ls` on the command line, the shell looks for a program called `ls` in a few designated places. Once if finds the program, it executes it.
- Most programs work with standard character-based input sources and output sinks — 'standard input', 'standard output' and 'standard error'. These are 'pipes' and can be connected to files, the terminal window, other programs, network feeds, etc. By default,
  - standard input ("stdin") is connected to your keyboard.
  - standard output and error ("stdout", "stderr") are connected to your console/terminal window.
- Most of these programs do something simple but very well — to do complex things, you string programs together.

Example: Practical I

1. Download the file `sumofhellos.c`) attached to this practical.

2. Modify line 6 (#define USERNAME "username") to replace username with your TCD username.

3. Compile the code - warnings are OK, but there should be no errors:
   ```
   cc -o sum sumofhellos.c -pthread
   ```

4. Run the executable (your username should appear seven times, as well as some number between 1 and 6 at the end):
   ```
   ./sum
   ```

5. Obtain a log of the compilation and running of the program by either:
   1. selecting it from your terminal window, copying it, and pasting in a text file called `practical1.log`.
   2. Or running the command
      ```
      ./sum > practical1.log
      ```

6. Bundle both your source code and logfile into a "tar" archive
   ```
   tar -r -f P1.tar sumofhellos.c practical1.log
   ```
   This will create a file called `P1.tar`.

7. Submit file `P1.tar` through Blackboard by the deadline.

Follow these steps precisely! This is really what is being assessed.

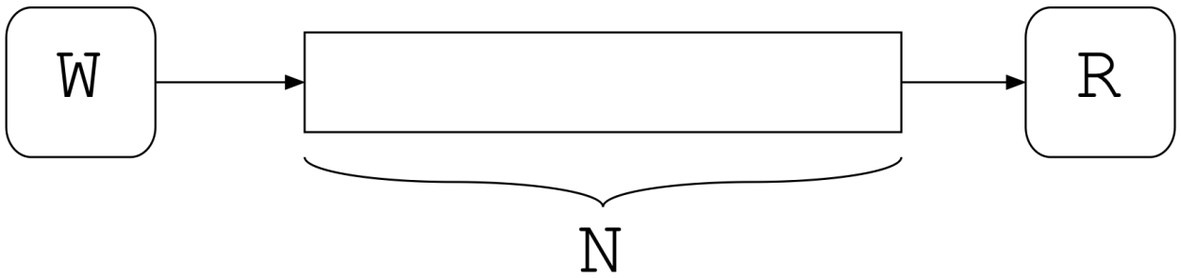
Combining commands

- Multiple commands:
  - Issue a sequence of commands on one line by separating them with a semicolon (“ **;** ”).
- Piping:
  - Using the bar symbol (“ | ”), you can direct (“pipe”) the standard output of one command into the standard input of the next one.
- Redirection:
  - You can redirect standard input to accept the contents of a file using symbol "<", and standard output to go to a file using symbol ">".
- Automation:
  - You can write a text file containing sequence of commands and shell commands and constructs. This can be executed as a *shell script*.

Producers and Consumers

- Essentially a *pipeline* of separate sequential programs.
  - E.g. concatenation of unix commands.
- Programs communicate via buffers.
  - Implemented in different ways, but, e.g.
    - Shared memory, flags, semaphores, etc.
    - Message passing over a network
- Flow of data is essentially one-way.

Imagine a Situation…



- We have a buffer that holds N items, with a counter recording how many items present.
- A 'Producer' writes (W) items to the buffer from time to time, incrementing its counter.
  - if the buffer is full, then the Producer should wait until room is freed up.
- A 'Consumer' reads (R) items from the buffer now and then, decrementing its counter.
  - if the buffer is empty, the Consumer should wait for a new item to arrive
- We could use a mutex to control access to the buffer & counter.

Write/Read behaviour (pseudo-code)

- Write(item,buffer) should do the following:
  - `if full(buffer) then wait_until_room(buffer);`
    `insert(item,buffer).`
- item = Read(buffer) should do the following:
  - `if empty(buffer) the wait_until_something(buffer);`
    `item = extract(buffer).`
- How do we use mutexes here?

Write/Read behaviour (pseudo-code)

- What's wrong with the following?:

  - ```
    lock(mutex);
    if full(buffer) then wait_until_room(buffer);
    insert(item,buffer);
    unlock(mutex).
    ```

  - ```
    lock(mutex);
    if empty(buffer) the wait_until_something(buffer);
    item = extract(buffer);
    unlock(mutex).
    ```

Both will wait forever, because, holding a lock on the mutex **while waiting,** prevents the other from ever getting the lock.

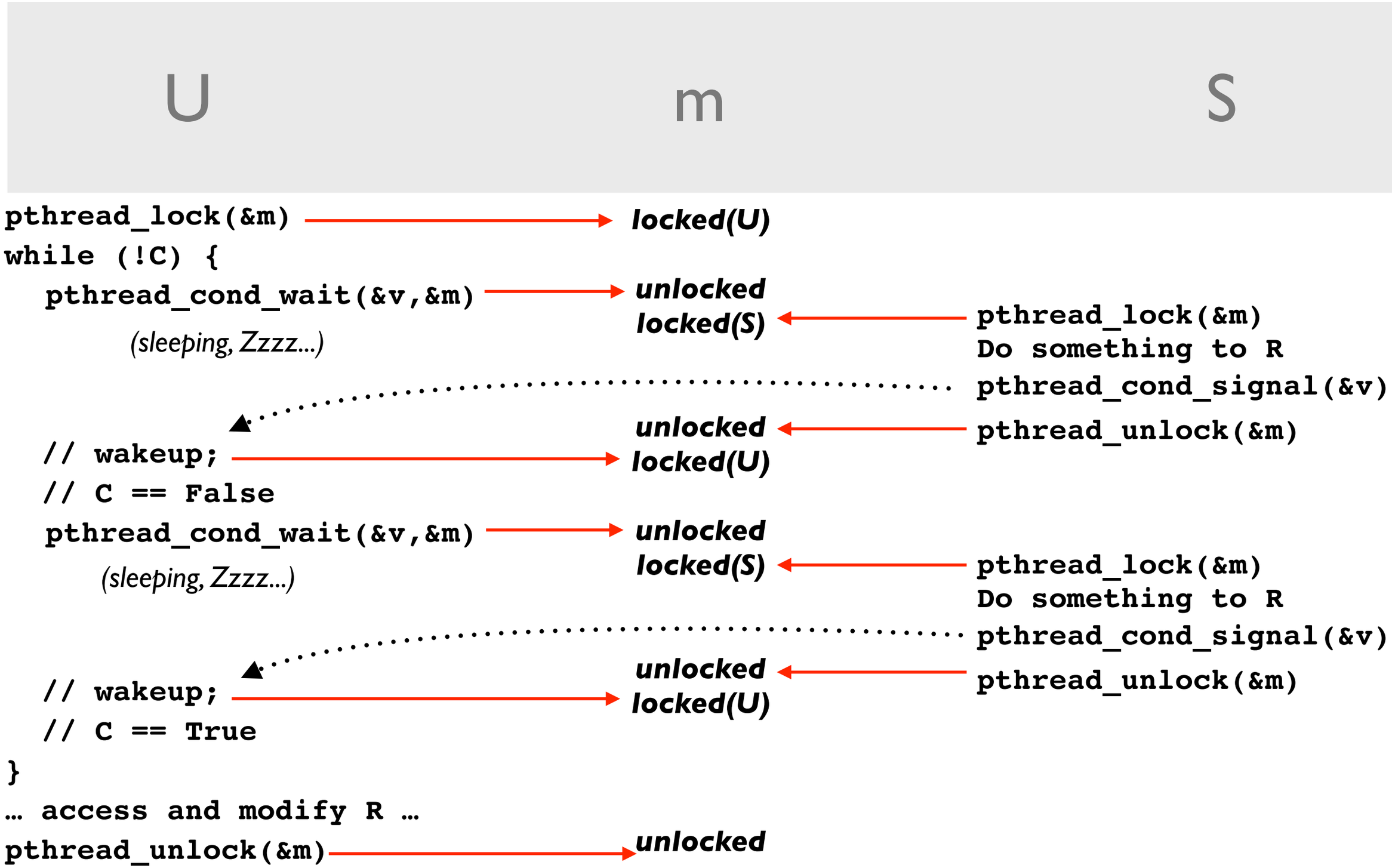
(A situation known as "DEADLOCK" !)

Write/Read behaviour

- We need to be able to:
  - wait for a state-change in global shared state (our buffer+counter),
  - without holding the lock all the time.
- The following does give the Consumer a chance to grab the lock. Will this work?
  -
```
wait = true;
while wait do
  lock(mutex); wait=full(buffer); unlock(mutex)
endwhile;
lock(mutex); insert(item,buffer); unlock(mutex).
```

Here the Producer is rapidly locking, checking, unlocking
- the Consumer risks "STARVATION" !

Mutexes aren't enough!

- Mutexes by themselves prevent efficient solutions
  - particularly when waiting for a resource to be in some specific state before proceeding with an operation on that resource.
- One solution to the previous problem is to require the waiting thread to sleep for a time interval before checking again:
  - ```
    wait = true;
    while wait do
      lock(mutex); wait=full(buffer); unlock(mutex);
      if (wait) then sleep(100);
    endwhile;
    lock(mutex); insert(item,buffer); unlock(mutex).
    ```
- The problem here is it is hard to find an optimal value for the sleep interval
  - why 100 here? why not 50? why not 500?, why not 5 ?(what units of time are they anyway?!)
    ```

Condition Variables

- A generalised synchronisation construct.
- Allows you to acquire mutex lock when a *condition* relying on a shared variable is true and to sleep otherwise.
- The ‘condition variable’ is used to manage the mutex lock and the signalling.
- Slightly tricky.

Condition Variables - Dramatis Personae

- A mutex variable M,
- A shared resource R to be guarded by M,
- A condition variable V,
- A condition C,
- A thread U that wishes to use R, protected by M, on condition that C is true,
- A thread S that will signal V, presumably when it has done something that might indirectly change the value of C.

Condition Variables vs. Conditions

- We need to be careful here to distinguish between "Conditions" and "Condition-Variables".
  - If not, much confusion will ensue
- Condition C
  - This is an application dependent true/false statement about something we want to be true in order to proceed
    - e.g. buffer is not full
- Condition-Variable V
  - this is a complex datastructure provided by a thread library to manage the mutexes and signalling required

From Thread U's POV (1)

- Acquire Mutex M
  - This controls access to R,
  - but also must control access to any shared variables that C depends on.
- if C is true, the mutex can be used as normal.
  - so proceed to access R as planned
- if C is false…

From Thread U's POV (2)

- If C is false, wait for condition variable V to be *signalled*.
- This is tricky, as access to R is controlled by M.
  - So, Mutex M is unlocked,
  - Thread sleeps, to be woken when V is signalled,
  - Then, M is re-acquired.
    - No guarantee you'll get it right away–maybe another thread will.
- So now, if V has been signalled…

From Thread U's POV (3)

- Since V has been signalled, there is a chance that the condition C is now true.
  - but another thread may have slipped in and done something that makes it false again!
- Re-evaluate C:
  - If true, then the mutex is available for you to use,
  - If false, back to previous page.
- Our 'if' needs to be replaced by a 'while'…

From Thread U's POV (4)

- Acquire Mutex M
- While !C
  - Unlock M
  - Wait for V to be signalled
  - Lock M
- process R as planned
- Unlock M (finished)

```
pthread_lock(&m)
while (!C) {
   // m is locked by U here
   pthread_cond_wait(&v,&m)
   // m is locked by U here
}
… access and modify R …
pthread_unlock(&m)
```

Thread S (1)

- Thread U is the 'user' of the condition variable V.
- If the condition is not true, U unlocks V's mutex M and sleeps, waiting for some other thread S to signal V and thereby waking U to check the condition again.

Thread S (2)

- Acquire Mutex M

```
pthread_lock(&m)
```

- Do something that affects R (and indirectly C)

```
Do something to R
```

- Signal the condition variable

```
pthread_cond_signal(&v)
```

  - This will awaken a thread that is waiting on the condition variable.

- Unlock Mutex M

```
pthread_unlock(&m)
```

My head hurts !

- Pthreads are not simple, nor straightforward
- There are (literally) a ***lot*** of moving parts!
- There are some key concepts that we need to understand
- The Lawrence Livermore tutorial gives a comprehensive overview
  - linked to from Blackboard, in the "READING/POSIX Threads" content area

“Formalising” Pthreads

- It will help to have a “formal” description of the PThread API calls we use.
- Why “formal” and not formal?
  - A Formal approach would use some form of mathematical logic to give a precise, unambiguous description of the API behaviour and usage.
    - We won’t go that far!
  - This “Formal” approach will state the properties needed using careful natural language

Properties of Interest

- Prototypes: types of values passed to and returned from API calls
  - Sometimes referred to as "(type-)signatures".
- Pre-Conditions: what needs to be true when we call an API function
- Post-Conditions: what will be true when we return from an API call
  - Provided the Pre-Condition was satisfied when it was called (!!)
- Invariants: what should always be true.
- In what follows,
  - we describe properties observed by the thread that calls the pthread function under discussion

pthread_mutex_init()

- Prototype: `int pthread_mutex_init( pthread_mutex_t *`
  `, const pthread_mutexattr_t * );`
- Call: `rc = pthread_mutex_init( &mymutex, NULL );`
- Precondition: None.
- Postcondition: `mymutex` is initialised, and unlocked.
- Invariant: all mutex variables (should be) initialised before use

pthread_mutex_lock()

- Prototype: `int pthread_mutex_lock( pthread_mutex_t * );`
- Call: `rc = pthread_mutex_lock( &mymutex );`
- Precondition: `mymutex` was initialised.
- Postcondition: `mymutex` is locked, and the calling thread "owns" that lock.
  - time may have passed if another thread owned the lock when this thread made the call.
- Invariant:
  - When a mutex is unlocked, no thread "owns" it.
  - When a mutex is locked, exactly one thread "owns" it.

pthread_mutex_unlock()

- Prototype: `int pthread_mutex_unlock( pthread_mutex_t * );`
- Call: `rc = pthread_mutex_unlock( &mymutex );`
- Precondition: `mymutex` is locked and calling thread "owns" that lock
- Postcondition: this thread no longer "owns" the lock
- Invariant:
  - When a mutex is unlocked, no thread "owns" it.
  - When a mutex is locked, exactly one thread "owns" it.

pthread_cond_init()

- Prototype: `int pthread_cond_init( pthread_cond_t *, const pthread_condattr_t * );`
- Call: `rc = pthread_cond_init( &myconvar, NULL );`

- Precondition: None.
- Postcondition: `myconvar` is initialised, and has not been "signalled".
- Invariant: all condition variables (should be) initialised before use

pthread_cond_wait()

- Prototype:
```
int pthread_cond_wait( pthread_cond_t *
                     , pthread_mutex_t * );
```
- Call:
```
rc = pthread_cond_wait( &myconvar, &mymutex );
```
- Precondition:
  - `myconvar` is initialised, and `mymutex` is locked.
  - the calling thread "owns" the lock
- Postcondition: `mymutex` is locked, and the calling thread "owns" that lock.
  - time may have passed until another thread signalled to `myconvar`.
- Invariant: ?

pthread_cond_signal()

- Prototype: `int pthread_cond_signal( pthread_cond_t * );`
- Call: `rc = pthread_cond_signal( &myconvar );`
- Precondition:
  - a previous call to `pthread_cond_wait`, usually by some other thread, has associated `myconvar` with `mymutex`.
  - `mymutex` is locked and "owned" by the calling thread.
- Postcondition:
  - a signal has been sent to `myconvar` to tell it to wake a thread sleeping on. on it.
  - `mymutex` is still locked and "owned" by the calling thread.
- Invariant: ?

Condition Variable Timelines

- It can help to view timelines
- Next slide shows the following scenario:
  - U wants to update R when C is true
  - It grabs mutex M, checks C (false) and so does a Wait on V, linked to M
    - Wait unlocks M and sleeps...
  - S updates R (with proper M usage) and signals V
  - V locks M, wakes U, but C is still false, so U waits again
    - Wait unlocks M and sleeps...
  - S updates R again and signals V
  - V locks M, wakes U, and C is now true, so U proceeds to modify R and then unlocks M


U
m
S
pthread_lock(&m)
locked(U)
while (!C) {
pthread_cond_wait(&v,&m)
unlocked
locked(S)
pthread_lock(&m)
Do something to R
(sleeping, Zzzz...)
pthread_cond_signal(&v)
unlocked
pthread_unlock(&m)
// wakeup;
locked(U)
// C == False
pthread_cond_wait(&v,&m)
unlocked
locked(S)
pthread_lock(&m)
Do something to R
(sleeping, Zzzz...)
pthread_cond_signal(&v)
unlocked
pthread_unlock(&m)
// wakeup;
locked(U)
// C == True
}
… access and modify R …
pthread_unlock(&m)
unlocked

Using Multiple Mutexes

- A typical program may have many critical shared resources, each protected by its own mutex (e.g. `resource`*i* protected by mutex `m`*i*).
- A thread may want exclusive access to more than one such resource at a time.
- This should be easy:
  - ```
    pthread_mutex_lock(&m1);
    pthread_mutex_lock(&m2);
    … do stuff to resource1 and resource2 …
    pthread_mutex_unlock(&m1);
    pthread_mutex_unlock(&m2);
    ```
- Let's see….
    ```

2 Mutexes (Preamble)

We have two resources, one mutex each for protection.

```
int resource1, resource2 ; // Two global resources

pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER ; // Protects resource1
pthread_mutex_t m2 = PTHREAD_MUTEX_INITIALIZER ; // Protects resource2
```

2 Mutexes (GoingUp)

```
// Going Up: sets r1 = min(r1,r2), r2 = r1+r2
void *GoingUp(void *a) {

    int tmp;

    pthread_mutex_lock(&m1);
    pthread_mutex_lock(&m2);

    tmp = resource1 + resource2;
    if (resource2 < resource1) { resource1 = resource2 ;}
    resource2 = tmp;

    pthread_mutex_unlock(&m2);
    pthread_mutex_unlock(&m1);

    pthread_exit(NULL);
}
```

2 Mutexes (GoingDown)

```
// Going Down: sets r1 = max(r1,r2), r2 = max(r1,r2)-min(r1,r2)
void *GoingDown(void *a) {
    int tmp;

    pthread_mutex_lock(&m2);
    pthread_mutex_lock(&m1);

    if (resource2 < resource1) {
        resource2 = resource1 - resource2;
    } else {
        tmp = resource2 - resource1;
        resource1 = resource2;
        resource2 = tmp;
    }

    pthread_mutex_unlock(&m1);
    pthread_mutex_unlock(&m2);

    pthread_exit(NULL);
}
```

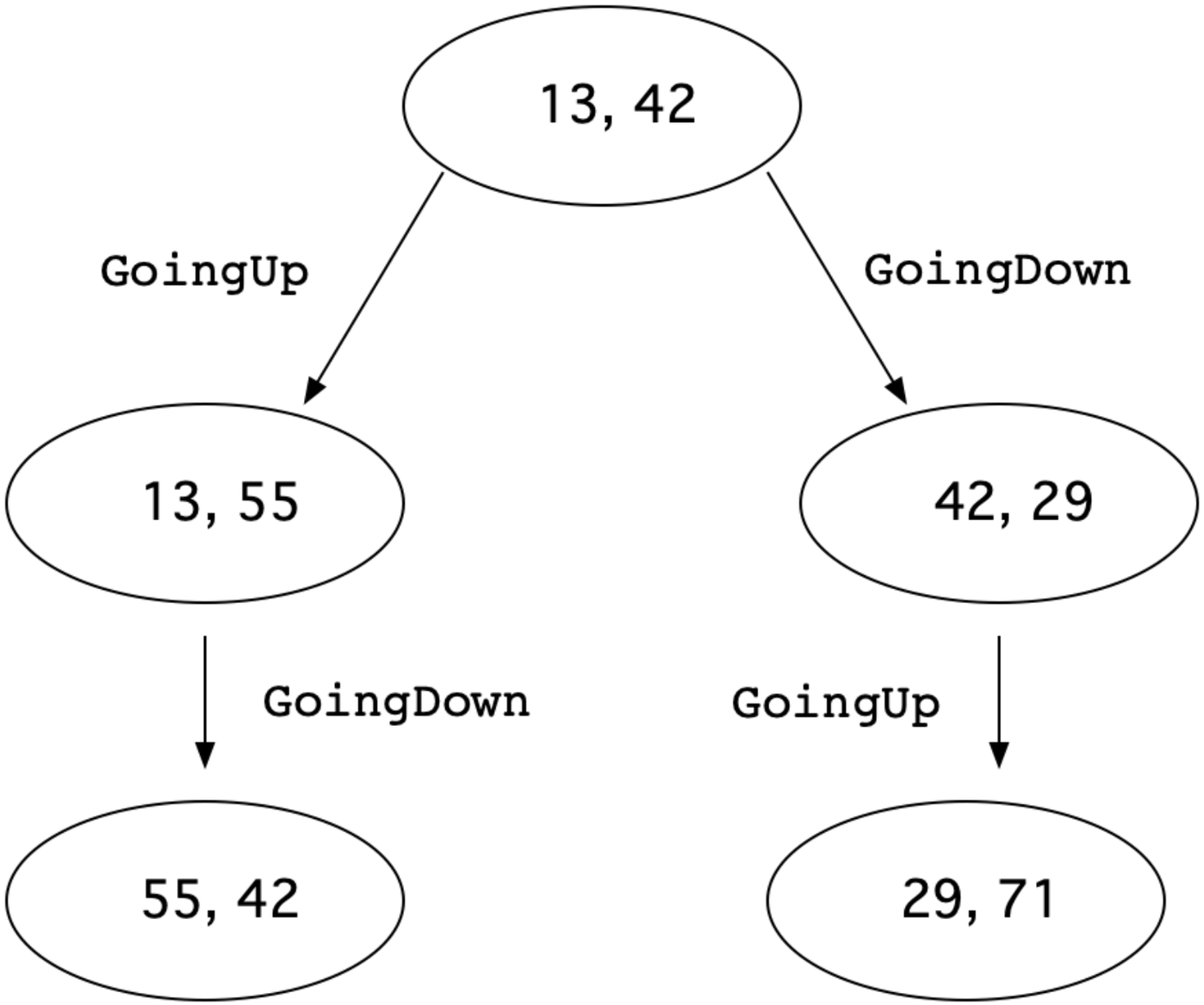
2 Mutexes (Main Program)

```
int main (int argc, const char * argv[]) {
    static pthread_t goingup,goingdown ;
    long rc;

    resource1 = 13; resource2 = 42;
    printf("r1,r2 = %d,%d\n", resource1, resource2);

    printf("Creating GoingUp:\n");
    rc = pthread_create(&goingup,NULL,GoingUp,(void *)0);
    if (rc) { ... }
    printf("Creating GoingDown:\n");
    rc = pthread_create(&goingdown,NULL,GoingDown,(void *)0);
    if (rc) { ... }

    printf("Waiting to join threads....\n");
    pthread_join( goingdown, NULL);
    pthread_join( goingup, NULL);

    printf("R1,R2 = %d,%d\n", resource1, resource2);
    printf("All Done!\n");

    return 0;
}
```

Expected Behaviour

- We have two threads, each of which (`GoingUp`,`GoingDown`) uses both resources.
  - r1,r2
- We expect to see one of two possibilities:
  - An execution of `GoingUp` followed by one of `GoingDown`.
    - 13,42 becomes 13,55 becomes 55,42
  - An execution of `GoingDown` followed by one of `GoingUp`.
    - 13,42 becomes 42, 29 becomes 29,71

Expected Behaviour (Diagram)



What actually happens!

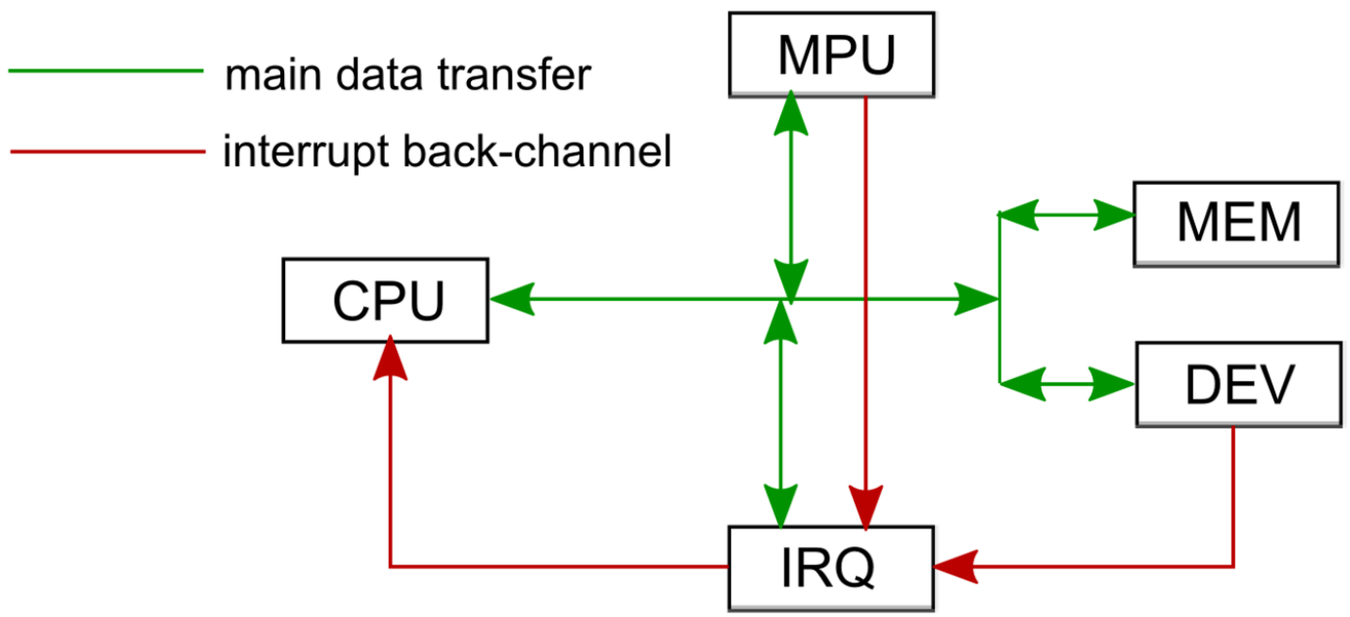
- Many times we see the expected behaviour
- But occasionally,
  - it hangs……..

- Another way to cause DEADLOCK !
- The solution:
  - all threads should acquire multiple locks in the same order as each other.

The life-time of a program

- A typical program alternates between times when it is:
  - Doing computation - lots of fast traffic between CPU and MEM
  - Doing Input/Output - short bursts of traffic between CPU and DEV with long waits in-between while **slow** I/O operations take place
- In early days, one program ran at a time
- The idea soon emerged to allow another program do its computation while the first one was waiting for I/O, and then another, and another, …
- The idea of a multi-user machine (a.k.a. "multiprogramming") was born
  - A concurrent system, requiring concurrent operating software.

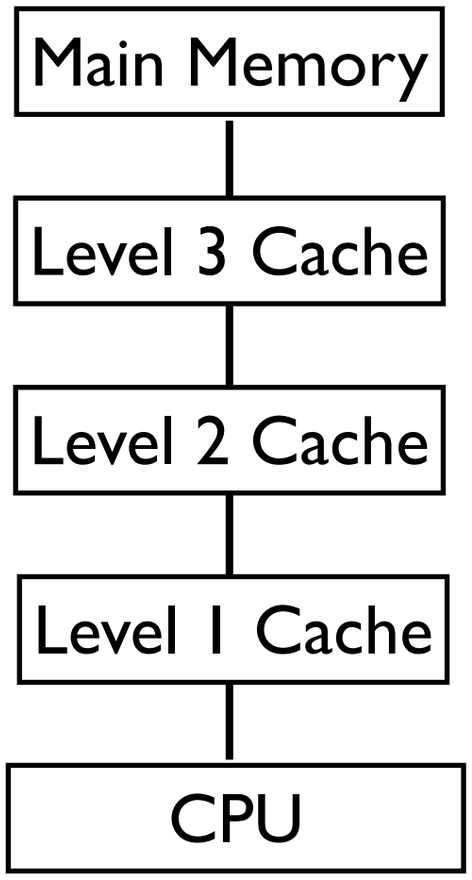
A Simple(?) Computer Architecture



For now, ignore the MPU and IRQ (we shall return to these)

CPU - central processing unit, executing instructions
MEM - fast memory, holding program and (live) data
DEV - hardware that interfaces to
(1) slow bulk storage devices (Hard Disks, etc
(2) outside world via I/O devices

Single Processor and Memory



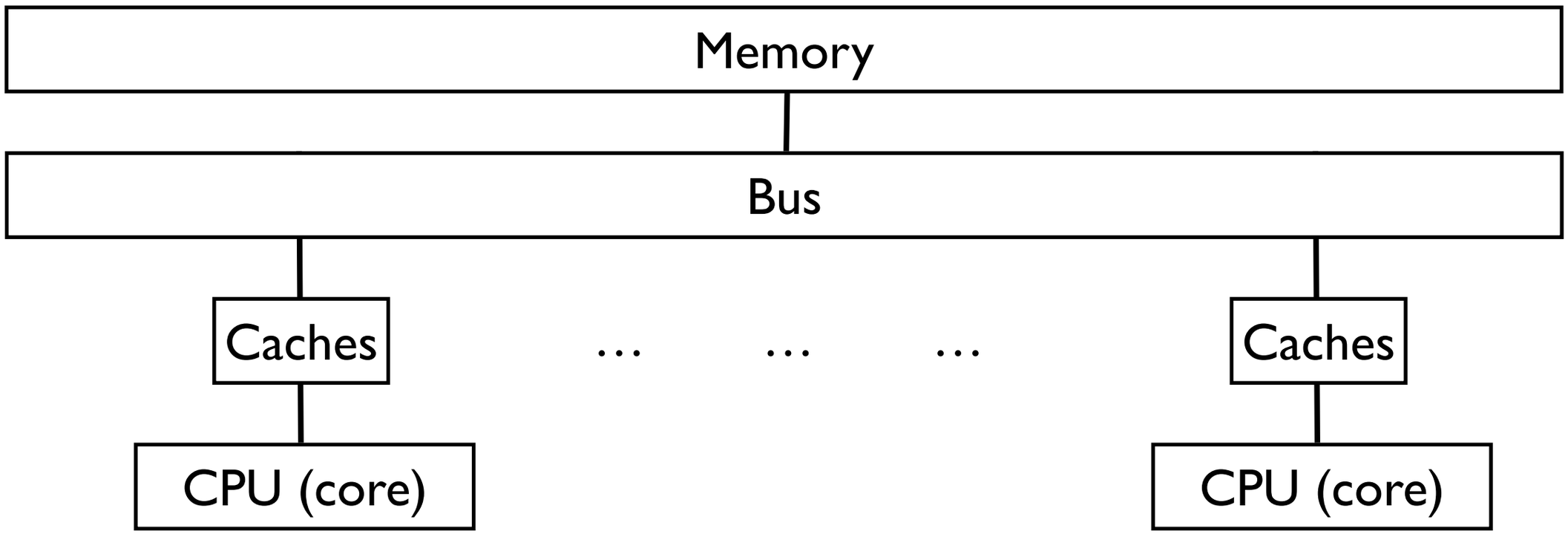
- Random Access Memory (RAM)
  - fairly slow, fairly cheap

- Hierarchy of caches between CPU and RAM.
  - Level 1 - smallest, fastest
  - ...
  - Level 3 largest, slowest.

- Register memory in the CPU:
  - very fast, very expensive
    - typically 400x faster than RAM !

(Part of) the so-called "Memory Hierarchy"

Multicore and Memory



Often the Level 3 cache is shared between all the CPUs

Shared Memory Machine

- Each processor has equal access to each main memory location – Uniform Memory Access (UMA).
  - Opposite: Non-Uniform Memory Access (NUMA)
- Each processor may have its own cache, or may share caches with others.
  - May have problems with cache issues, e.g. consistency, line-sharing, etc.

Sequential Program

- A ***Sequential Program*** has one site of program execution. At any time, there is only one site in the program where execution is under way.
- The ***Context*** of a sequential program is the set of all variables, processor registers (including hidden ones that can affect the program), stack values and, of course, the code, assumed to be fixed, that are current for the single site of execution.
- The combination of the context and the flow of program execution is often called [informally] a *thread*.

Concurrent Program

- A ***Concurrent*** *Program* has more than one site of execution. That is, if you took a snapshot of a concurrent program, you could understand it by examining the state of execution in a number of different sites in the program.
- Each site of execution has its own context—registers, stack values, etc.
- Thus, a concurrent program has *multiple threads* of execution.

Parallel Program

- A ***Parallel*** Program, like a concurrent program, has multiple threads of program execution.
- A key difference between *concurrent* and *parallel* is
  - In concurrent programs, only one execution agent is assumed to be active. Thus, while there are many execution sites, only one will be active at any time.
  - In parallel programs, multiple execution agents are assumed to be active simultaneously, so many execution sites will be active at any time.
- Another key difference:
  - Concurrency is a way of designing solution to problems by considering them to be composed of separate threads of execution that run independently, but are also able to interact when needed.
  - Parallelism is a way to speed up program execution when multiple computing cores are available by breaking a computation into pieces, each of which runs on its own core.

Interaction

- Most of the issues to do with threads (really, with any kind of concurrent/parallel processing) arise over unforeseen interaction sequences.
  - For example, if two threads attempt to increment the same variable

Unsafe Interaction Example

Thread 1	Thread 2	G
L=G (4)	–	4
L=L+10 (14)	–	4
	L=G (4)	4
G=L	L++ (5)	14
	G=L	5

G = Global Variable; L = Separate Local Variables

Interactions & Dependencies

- Interactions occur because of *dependencies*.
- Interactions can be made safe using a variety of techniques.
  - E.g. mutexes, condition variables, etc.
    - We have already had a look at them…
    - They tend to be expensive; sometimes *very* expensive.
- But, sometimes we can redesign the code to avoid dependencies.
- One of the biggest themes in this kind of programming is *dependency minimisation*.

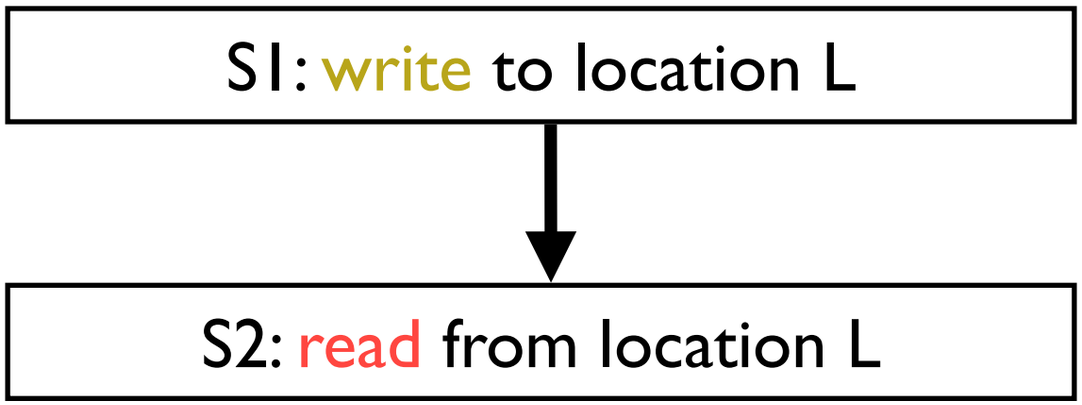
Why is it this difficult?

- It's not clear whether it *really* is more difficult to think about using multiple agents, e.g. multiple processors, to solve a problem:
  - Maybe we are conditioned into thinking about just one agent;
  - Maybe it's natural;
  - Maybe it's our notations and toolsets;
- Of course, maybe it really is trickier.
  - One possible culprit - often the simplest model we have of concurrent program behaviour is to assume that decision to switch between threads is purely non-deterministic (arbitrary, random).
    - This makes it hard to predict when things will happen.

Dependency

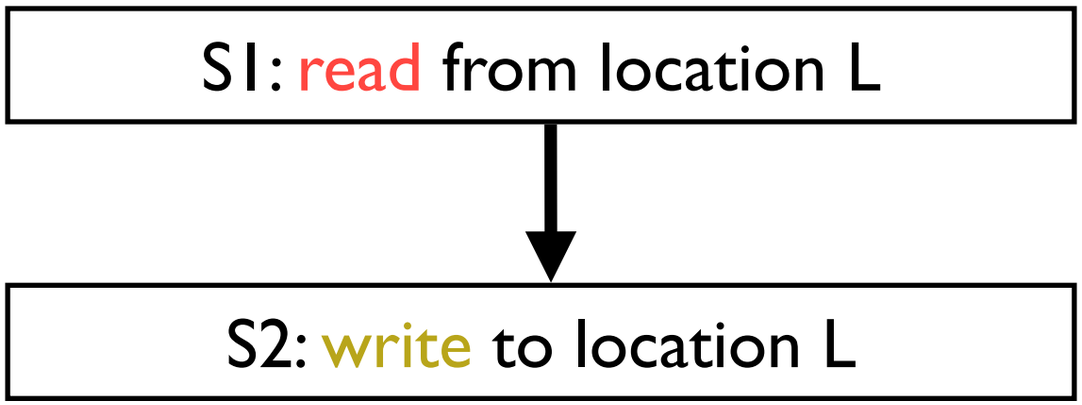
- Independent sections of code can run independently.
- We can analyse code for dependencies.
  - To preserve the meaning of the program;
  - To transform the program to reduce dependencies and improve parallelism.
- We typically define dependencies of various kinds between two program statements, (S1 & S2), that can be in the same thread, or in different ones.

1 – Flow Dependence



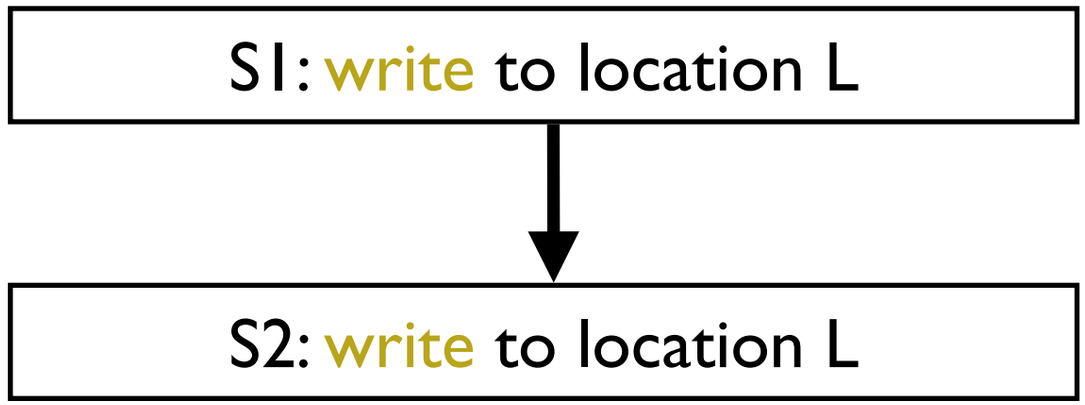
- Flow Dependence: S2 is flow dependent on S1 because S2 reads a location S1 writes to.
  - It must be written to (S1) before it's read from (S2)

2 – Antidependence



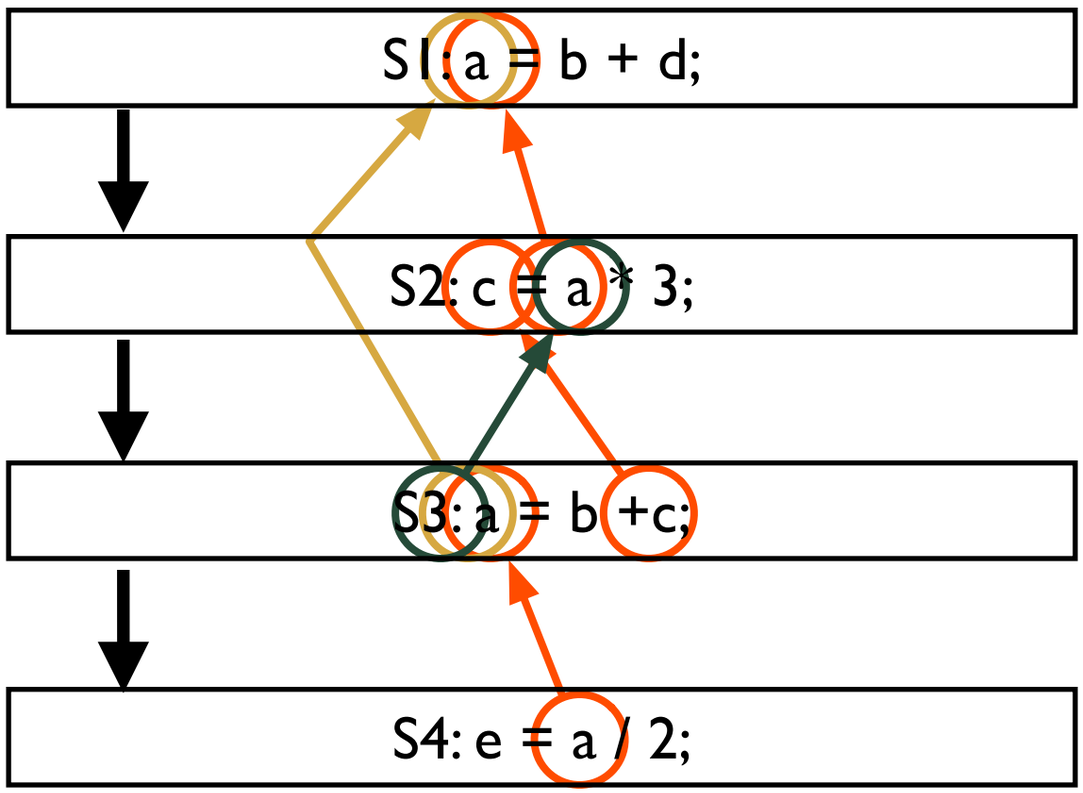
- Antidependence: S2 is antidependent on S1 because S2 writes to a location S1 reads from.
  - It must be read from (S1) before it can be written to (S2)

3 – Output Dependence



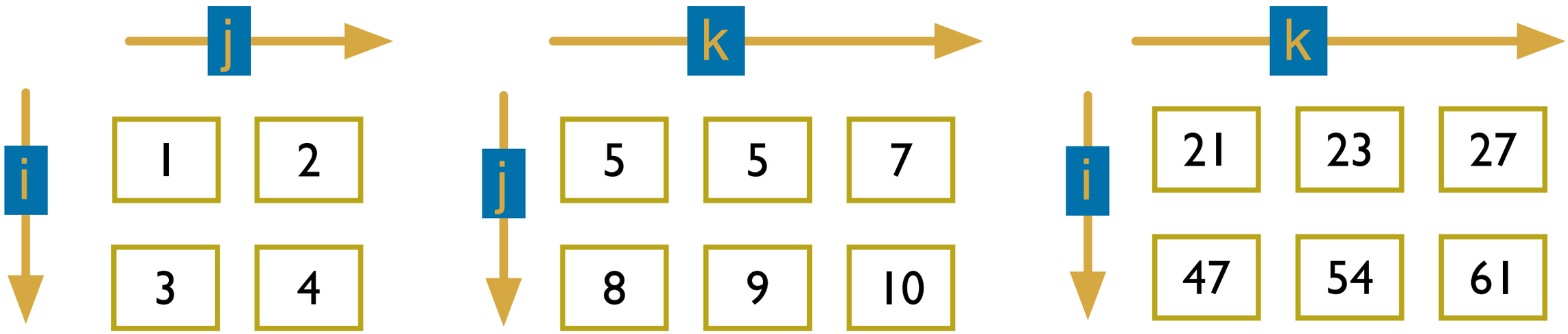
- Output dependence: S2 is output dependent on S1 because S2 writes to a location S1 writes to.
  - The value of L is affected by the order of S1 and S2.

Example



Removing Unneeded Dependencies in Loops

- Example. We know, intuitively, we can parallelise this a great deal:
  - each element in C is independent



A X B = C

Sample Serial Solution

```
for i = 1  to 2 {
  for k = 1 to 3 {
    sum = 0.0;
    for j = 1 to 2
      sum = sum + a[i,j] * b[j,k];
    c[i,k] = sum;
  }
}
```

How can we transform it?

- We can work out how to transform it by locating the dependencies in it.
- Some dependencies are intrinsic to the solution, but
- Some are *artefacts* of the *way* we are *solving* the problem;
  - If we can identify them, perhaps we can modify or remove them.

Try three execution agents:

```
with k = 1;
for i = 1  to 2 {
 for k = 1 to 3 {
    sum = 0.0;
   for j = 1 to 2
     sum = sum + a[1,j] * b[j,k];
   c[i,k] = sum;
 }
}
```

```
with k = 2;
for i = 1  to 2 {
 for k = 1 to 3 {
    sum = 0.0;
   for j = 1 to 2
     sum = sum + a[1,j] * b[j,k];
   c[i,k] = sum;
 }
}
```

```
with k = 3;
for i = 1  to 2 {
 for k = 1 to 3 {
    sum = 0.0;
   for j = 1 to 2
     sum = sum + a[1,j] * b[j,k];
   c[i,k] = sum;
}
}
```

Issues:

- The variable ***sum***, as written, is common to all three programs.
- Solution:
  - Make ***sum*** private to each program to avoid this dependency.

Try Six Execution Agents

```
with k = 1, i=1;
for i = 1 to 2 {
 for k = 1 to 3 {
    sum = 0.0;
    for j = 1 to 2
      sum = sum + a[1,j] * b[j,k];
    c[i,k] = sum;
 }
}
```

```
with k = 2, i=1;
for i = 1 to 2 {
 for k = 1 to 3 {
    sum = 0.0;
    for j = 1 to 2
      sum = sum + a[1,j] * b[j,k];
    c[i,k] = sum;
 }
}
```

```
with k = 3, i=1;
for i = 1 to 2 {
 for k = 1 to 3 {
    sum = 0.0;
    for j = 1 to 2
      sum = sum + a[1,j] * b[j,k];
    c[i,k] = sum;
 }
}
```

```
with k = 1, i=2;
for i = 1 to 2 {
 for k = 1 to 3 {
    sum = 0.0;
    for j = 1 to 2
      sum = sum + a[1,j] * b[j,k];
    c[i,k] = sum;
 }
}
```

```
with k = 2, i=2;
for i = 1 to 2 {
 for k = 1 to 3 {
    sum = 0.0;
    for j = 1 to 2
      sum = sum + a[1,j] * b[j,k];
    c[i,k] = sum;
 }
}
```

```
with k = 3, i=2;
for i = 1 to 2 {
 for k = 1 to 3 {
    sum = 0.0;
    for j = 1 to 2
      sum = sum + a[1,j] * b[j,k];
    c[i,k] = sum;
 }
}
```

Summarising:

- We could parallelise the original algorithm with some care:
  - Private Variables to avoid unnecessary dependencies
    - But we may need to combine private results at the end to get a global answer.
- Actually, we could break this ‘Embarrassingly Parallel’ problem into tiny separate pieces; maybe too small. *(How will we know?)*

How fast can we go?

- We have a sequential program that runs too slow
- We have extra hardware resources that could be used to speed it up.
- How fast can we go?

Do the math ….

$T(n)$	Time to run program with n parallel processors
T	Shorthand for $T(1)$
$S(n)$	Speedup with n processors
$E(n)$	Efficiency of Speedup
p	Proportion of T spent executing parallelisable part.
s	Speedup possible for parallelisable part

Speedup and Efficiency

Maximum speedup:

$$T(n) \geq T(1)/n$$

Speedup:

$$S(n) = T(1)/T(n)$$

Efficiency:

$$E(n) = S(n)/n$$

Program Time and Effective Speedup

- Time to run program without parallelism:

$$T = (1 - p)T + pT$$

- Parallel (effective) speedup vs. processor count:

$$s \leq n$$

Speedup related to *n* and *s*.

Time to run program with speedup s of parallelisable part:

$$T(n) = (1-p)T + pT/s = (1-p+p/s)T$$

Speedup when running program with parallelisable speedup s:

$$\begin{aligned} S(n) &= T/(1-p+p/s)T \\ &= 1/(1-p+p/s) \end{aligned}$$

Amdahl's Law

$$S(n) = \frac{1}{1 - p + (p/s)} \qquad s \leq n$$

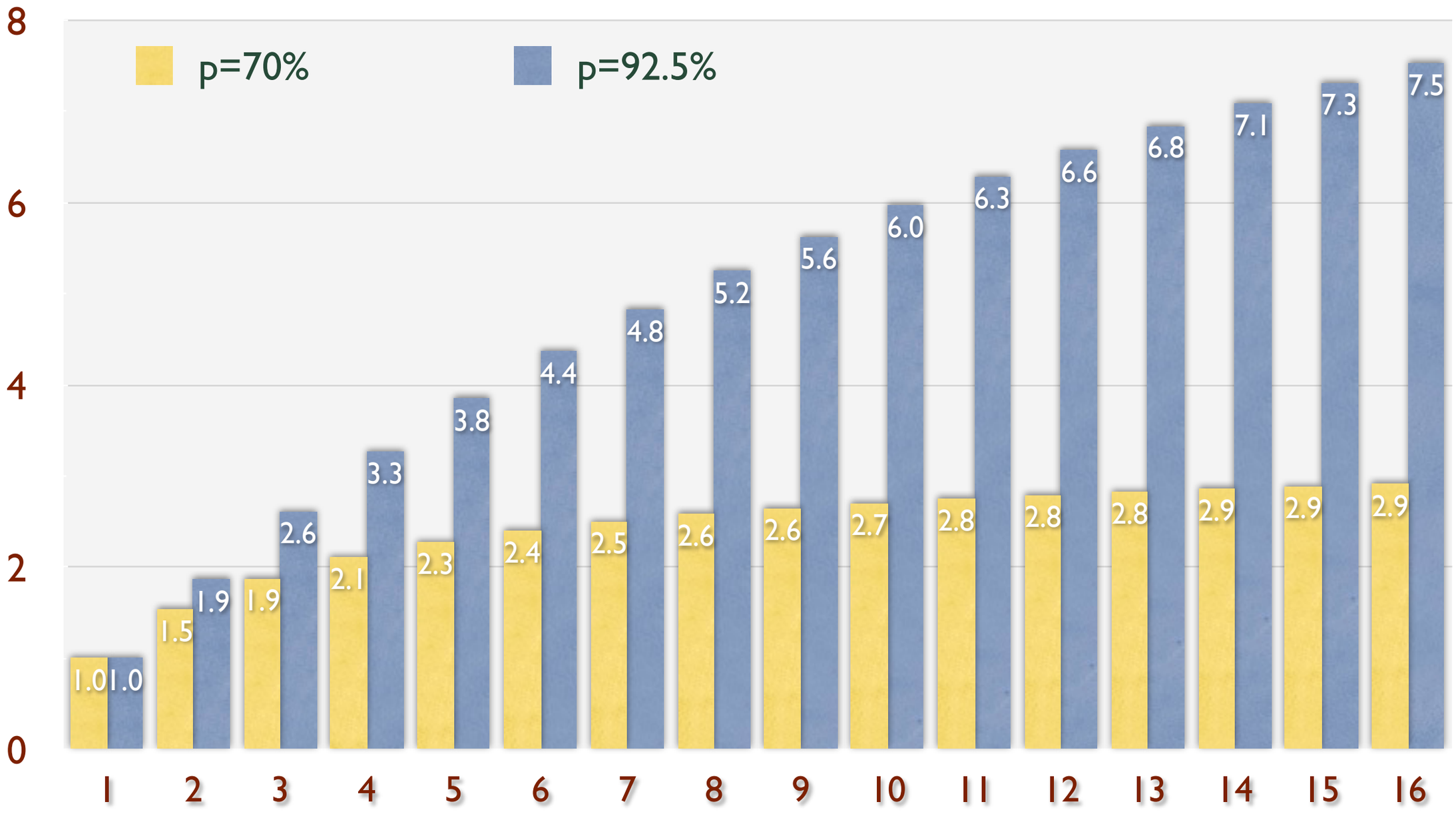
p - proportion of single processor time we can parallelise

n - number of parallel processors

s - parallel speedup ratio

$S(n)$ - overall program speedup

Graphically, Bad News



Amdahl's Law as n (and s) get large

$$S(n) = \frac{1}{1 - p + (p/s)} \quad s \leq n$$

Limit as n, s get very large, so $p/s \to 0$:

$$\frac{1}{1 - p}$$

p=0.925, S(n) ⟶ 13.333... ; *p=0.75, S(n) ⟶ 4*

Implications

- Even a small fraction of sequential code in a program can seriously interfere with speedup.
  - Note that the code protected by a mutex can only run sequentially!
  - If code has wait a while for a mutex, then that waiting time, has to be added in.
- To maximise performance, inherently sequential code has to be minimised.

Remember Integration with mutexes?

```
pthread_mutex_t  mutex = PTHREAD_MUTEX_INITIALIZER;

void *IntegratePart(void *i) {

  double a,b,area;
  int rc;

  a = (int)i * H ;
  b = a + H;
  area = trapezoid(a,b);

  // critical section with mutex
  rc = pthread_mutex_lock(&mutex);
    checkResults("pthread_mutex_lock()\n", rc);

  answer=answer+area;

  rc = pthread_mutex_unlock(&mutex);
    checkResults("pthread_mutex_lock()\n",rc);

    pthread_exit(NULL);
}
```

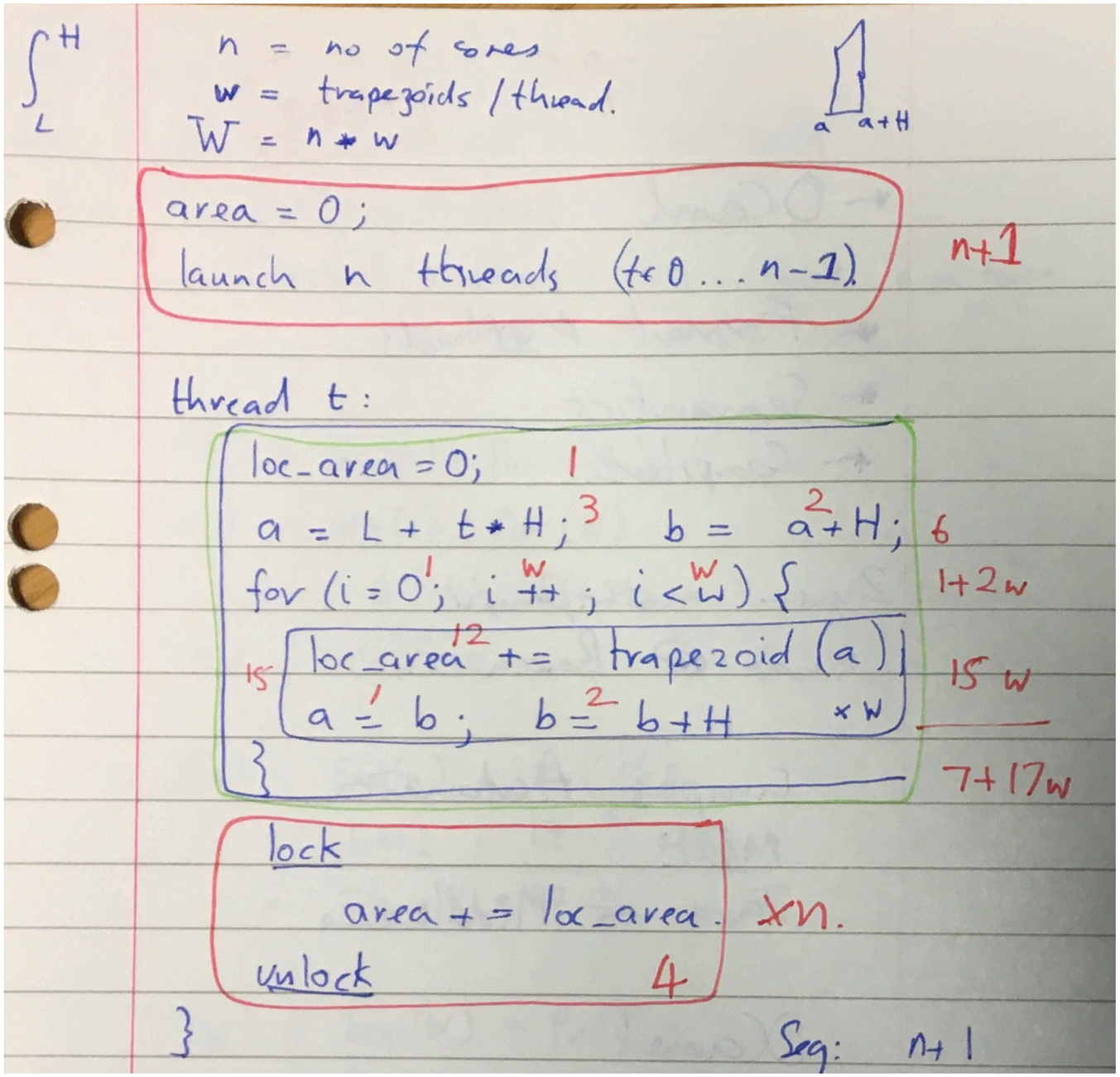
What does Amdahl's law say here?

Assume we have one processor per slice?

Some (simplifiying) assumptions.

- Assume
  - we can ignore the time spent creating threads
  - assume mutex lock and unlock each require one time-step
  - assume assignment and each arithmetic operation or function call require one time step.:
- Parallelisable code computes a, b, and area (using `trapezoid`, which uses `f`)
  - Cost of `f` is 3
  - Cost of `trapezoid` is 10
- Total cost for a, b and area is 15.
- Serial Code is mutex lock/unlock and answer update: total cost is 4.

Using a thread to do several trapezoids



We get two formulas

Sequential Part:
Seq(n) = 5n+1

Parallel Part:
Par(w) = 17w+7

Time in terms of n, if W=1000:

T(n) = 5n + 8 + 17000/n

$T(n) = 5n + 8 + 17000/n$

- A bit of calculus shows T(n) a minimum for n = 92, when it equals 653 steps
  - T(1) = 17013
  - T(92) = 653
  - T(1000) = 5025
  - T(3400) = 17013
- So throwing 3400 processors at this problem is as slow as using one!

Revisiting Matrix Multiplication

(keeping Amdahl's law in mind)

```
for i = 1  to 2 {
  for k = 1 to 3 {
    sum = 0.0;
    for j = 1 to 2
      sum = sum + a[i,j] * b[j,k];
    c[i,k] = sum;
  }
}
```

What is `p`, the proportion of this program that can be sped-up ?

We need to consider a specific strategy...

e.g., parallelising just the `k` loop:

```
with k = 1;
for i = 1  to 2 {
 for k = 1 to 3 {
    sum = 0.0;
    for j = 1 to 2
      sum = sum + a[1,j] * b[j,1];
    c[i,1] = sum;
 }
}
```

```
with k = 2;
for i = 1  to 2 {
 for k = 1 to 3 {
    sum = 0.0;
    for j = 1 to 2
      sum = sum + a[1,j] * b[j,2];
    c[i,2] = sum;
 }
}
```

```
with k = 3;
for i = 1  to 2 {
 for k = 1 to 3 {
    sum = 0.0;
    for j = 1 to 2
      sum = sum + a[1,j] * b[j,3];
    c[i,3] = sum;
}
}
```

What is `p`, the proportion of this program that can be sped-up ?

How do we handle updating `sum`?
Make it a critical resource? Local copies?

More specific...

parallelising just the `k` loop, with local sums

```
with k = 1;
for i = 1  to 2 {
 for k = 1 to 3 {
    sum[1] = 0.0;
    for j = 1 to 2
      sum[1] = sum[1] + a[1,j] * b[j,1];
    c[i,1] = sum[1];
 }
}
```

```
with k = 2;
for i = 1  to 2 {
 for k = 1 to 3 {
    sum[2] = 0.0;
    for j = 1 to 2
      sum[2] = sum[2] + a[1,j] * b[j,2];
    c[i,2] = sum[2];
 }
}
```

```
with k = 3;
for i = 1  to 2 {
 for k = 1 to 3 {
    sum[3] = 0.0;
    for j = 1 to 2
      sum[3] = sum[3] + a[1,j] * b[j,3];
    c[i,3] = sum[3];
}
}
```

What is `p`, the proportion of this program that can be sped-up ?

Each `sum[k]` is only used locally!

Try Six Execution Agents again

```
with k = 1, i=1;
for i = 1 to 2 {
 for k = 1 to 3 {
    sum[1,1] = 0.0;
    for j = 1 to 2
      sum[1,1] = sum[1,1] + a[1,j] * b[j,1];
    c[1,1] = sum[1,1];
 }
}
```

...

```
with k = 3, i=1;
for i = 1 to 2 {
 for k = 1 to 3 {
    sum[1,3] = 0.0;
    for j = 1 to 2
      sum[1,3] = sum[1,3] + a[1,j] * b[j,3];
    c[1,3] = sum[1,3];
 }
}
```

`sum` is redundant now, and we can simply use`c` itself to compute the result.

```
with k = 1, i=2;
for i = 1 to 2 {
 for k = 1 to 3 {
    sum[2,1] = 0.0;
    for j = 1 to 2
      sum[2,1] = sum[2,1] + a[2,j] * b[j,1];
    c[2,1] = sum[2,1];
 }
}
```

...

```
with k = 3, i=2;
for i = 1 to 2 {
 for k = 1 to 3 {
    sum[2,3] = 0.0;
    for j = 1 to 2
      sum[2,3] = sum[2,3] + a[2,j] * b[j,3];
    c[2,3] = sum[2,3];
 }
}
```

Try Six Execution Agents again

```
with k = 1, i=1;
for i = 1 to 2 {
 for k = 1 to 3 {
    c[1,1] = 0.0;
    for j = 1 to 2
       c[1,1] = c[1,1] + a[1,j] * b[j,1];
}
}
```

...

```
with k = 3, i=1;
for i = 1 to 2 {
 for k = 1 to 3 {
    c[1,3] = 0.0;
    for j = 1 to 2
      c[1,3] = c[1,3] + a[1,j] * b[j,3];
}
}
```

```
with k = 1, i=2;
for i = 1 to 2 {
 for k = 1 to 3 {
    c[2,1] = 0.0;
    for j = 1 to 2
       c[2,1] = c[2,1] + a[2,j] * b[j,1];
}
}
```

...

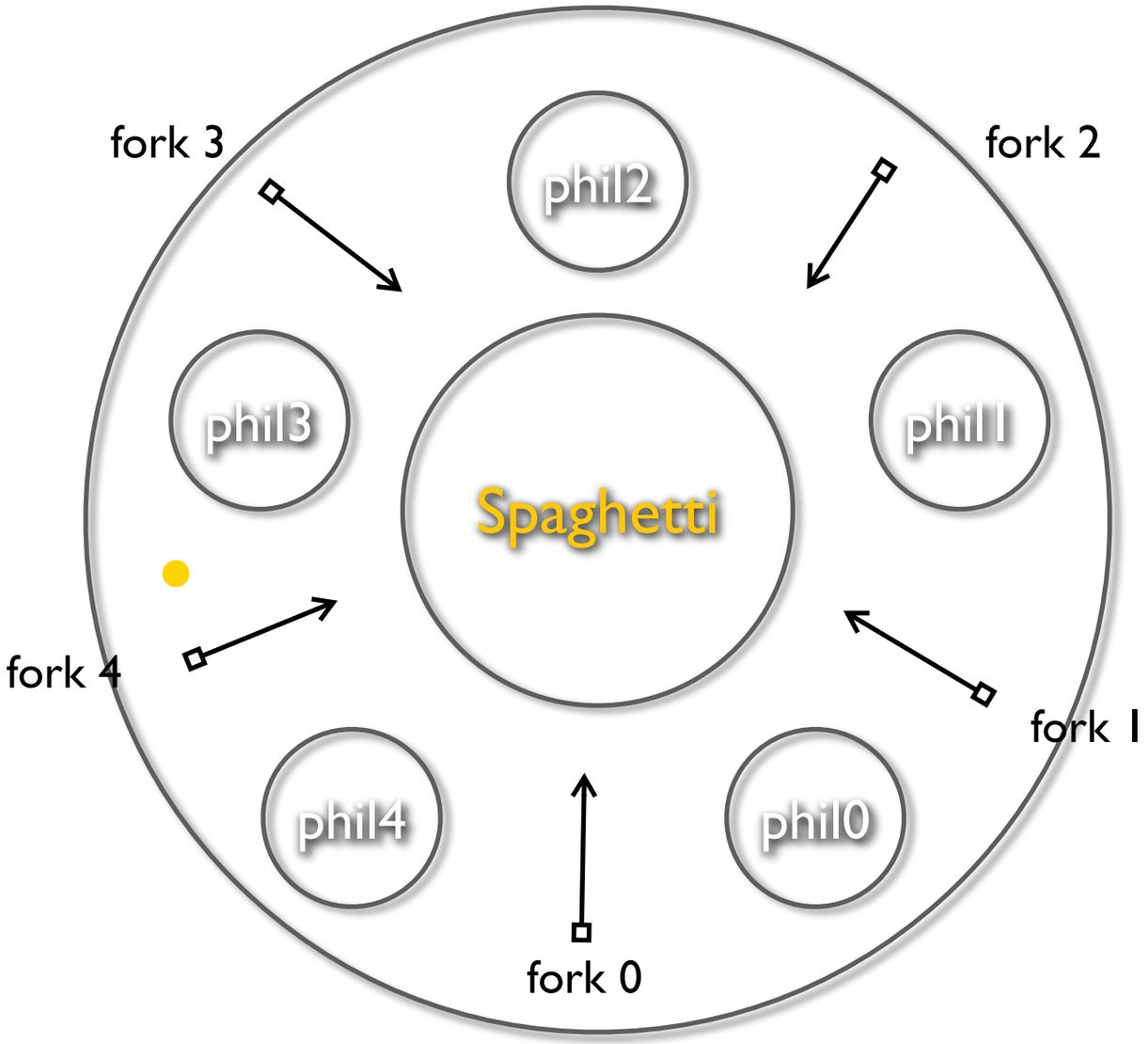
```
with k = 3, i=2;
for i = 1 to 2 {
 for k = 1 to 3 {
    c[2,3] = 0.0;
    for j = 1 to 2
      c[2,3] = c[2,3] + a[2,j] * b[j,3];
}
}
```

Here we see we have full output independence between each component of `c`.

Matrix Multiply is easy (?)

- Matrix multiply turns out to be highly ("embarrasingly", "massively") parallelisable!
- In principle we should be able to get `p` very close to 1.
  - Indeed many supercomputing facilities use massive parallelism to do matrix calculations
- However, for a M x M (square) matrix, the equivalent of the "Six Execution Agents" solution requires M^2 processors!
- Also, there are always hidden "non-p" costs in practice
  - Getting fast access by N cores to main memory gets harder to achieve as N grows large
  - While each location in matrix C is only written by one core, each location in both A and B are read by many. Again, this can lead to "bus contention" as processors wait to read shared memory.
    - (Clever understanding of cache behaviour can lead to algorithms that minimise such contention)

Dining Philosophers Problem



Philosophers want to
repeatedly
think and then eat.
Each needs two forks.
Infinite supply of pasta (ugh!).
How to ensure they don't starve.

A model problem for thinking about
problems in Concurrent Programming:
Deadlock
Livelock
Starvation/Fairness
Data Corruption

Dining Philosophers

- Features:
  - A philosopher eats only if they have two forks.
  - No two philosophers may hold the same fork simultaneously
- Characteristics of Desired Solution:
  - Freedom from deadlock
  - Freedom from starvation
  - Efficient behaviour generally.

Modelling the Dining Philosphers

- We imagine the philosophers participate in the following observable events:

Event Shorthand	Description
think.p	Philosopher *p* is thinking.
eat.p	Philosopher *p* is eating
pick.p.f	Philosopher *p* has picked up fork *f*.
drop.p.f	Philosopher *p* has dropped fork *f*.

What a philosopher does:

- A philosopher wants to: *think ; eat ; think ; eat; think ; eat ; ....*
- In fact each philosopher needs to do: *think ; pick forks ; eat ; drop forks ; ...*
- We can describe the behaviour of the *i*th philosopher as:

Phil(i) = think.i ; pick.i.i ; pick.i.i+ ; eat.i ; drop.i.i ; drop.i.i+ ; Phil(i)

  - Here *i+* is shorthand for *(i+1) mod 5*.

- What we have are five philosophers running in parallel (||):

Phil(0) || Phil(1) || Phil(2) || Phil(3) || Phil(4)

What can (possibly) go wrong ?

- Consider the following (possible, but maybe unlikely) sequence of events, assuming that, just before this point, all philosophers are *think.p*-ing…

  pick.0.0 ; pick.1.1 ; pick.2.2 ; pick.3.3 ; pick 4.4 ;

- At this point, every philosopher has picked up their left fork.
  - Now each of them wants to pick up its right one.
  - But its right fork is its righthand neighbours left-hand fork!
  - Every philosopher wants to *pick.i.i+* , but can't, because it has already been *pick.i+.i+*-ed!
  - Everyone is stuck and no further progress can be made
- DEADLOCK !

"Implementing" *pick* and *drop*

- In effect *pick.p.f* attempts to lock a mutex protecting fork *f*.
- So each philosopher is trying to lock two mutexes for two forks before they can *eat.p*.
- The *drop.p.f* simply unlocks the mutex protecting *f*.

You can't always rely on the scheduler…

- A possible sequence we might observe, starting from when philosophers 1 and 2 are thinking, could be:

  pick.1.1 ; pick.2.2 ; pick.2.3 ; eat.2 ; drop.2.2

  - now, philosopher 1 has picked fork 1 but is waiting for it to be dropped by philosopher 2.
  - But philosopher 2 is still running, and so drops the other fork, has a quick think, and then gets quickly back to eating once more:

    pick.1.1 ; pick.2.2 ; pick.2.3 ; eat.2 ; drop.2.2 ; drop.2.3 ; think.2 ; pick.2.2 ; …

  - Philosopher 1 could get really unlucky and never be scheduled to get the lock for fork 2. It is queuing on the mutex for fork 2, but when philosopher 2 unlocks it, somehow the lock, and control is not handed to philosopher 1.

- STARVATION (and its close friend UN-FAIRNESS)

Communication

- A concurrent or parallel program consists of two or more separate threads of execution, that run independently except when they have to interact
- To interact, they must *communicate*
- Communication is typically implemented by
  - sharing memory
    - One thread writes data to memory; the other reads it
  - passing messages
    - One thread sends a message; the other gets it

The Challenge of Concurrency

- Conventional testing and debugging is not generally useful.
  - We assume that the sequential parts of concurrent programs are correctly developed.
  - Beyond normal sequential-type bugs, there is a whole range of problems caused by errors in communication and synchronisation. They can not easily be reproduced and corrected.
  - So, we are going to use notions, algorithms and formalisms to help us design concurrent programs that are correct by design.

Sequential Process

- A *sequential process* is the execution of a sequence of *atomic statements*.
  - E.g. Process P could consist of the execution of $P_1, P_2, P_3, \ldots$.
  - Process Q could be $Q_1, Q_2, Q_3, \ldots$.
- We think of each sequential process as being a distinct entity that has its own separate program counter (PC).

Concurrent Execution

- A concurrent system is modelled as a collection of sequential processes, where the atomic statements of the sequential processes can be arbitrarily *interleaved*, but respecting the sequentiality of the atomic statements within their sequential processes.
- E.g. say P is P_1,P_2 and Q is Q_1,Q_2.

Scenarios for P and Q.

p1 → q1 → p2 → q2
p1 → q1 → q2 → p2
p1 → p2 → q1 → q2
q1 → p1 → q2 → p2
q1 → p1 → p2 → q2
q1 → q2 → p1 → p2

Notation

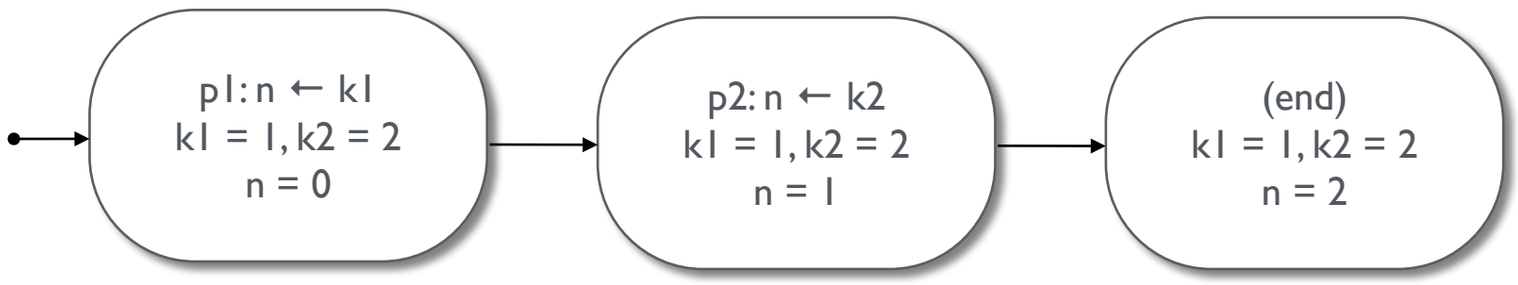
Trivial concurrent program: processes **p** and **q** each do one action that updates global n with the value of their local variable.

Trivial Concurrent Program (Title)	
integer n ←0 (Globals)	
p (Process Name)	**q**
integer k1 ← 1 (Locals)	integer k2 ← 2
p1: n ← k1 (Atomic Statements)	q1: n ← k2

All global and local variables are initialised when declared.

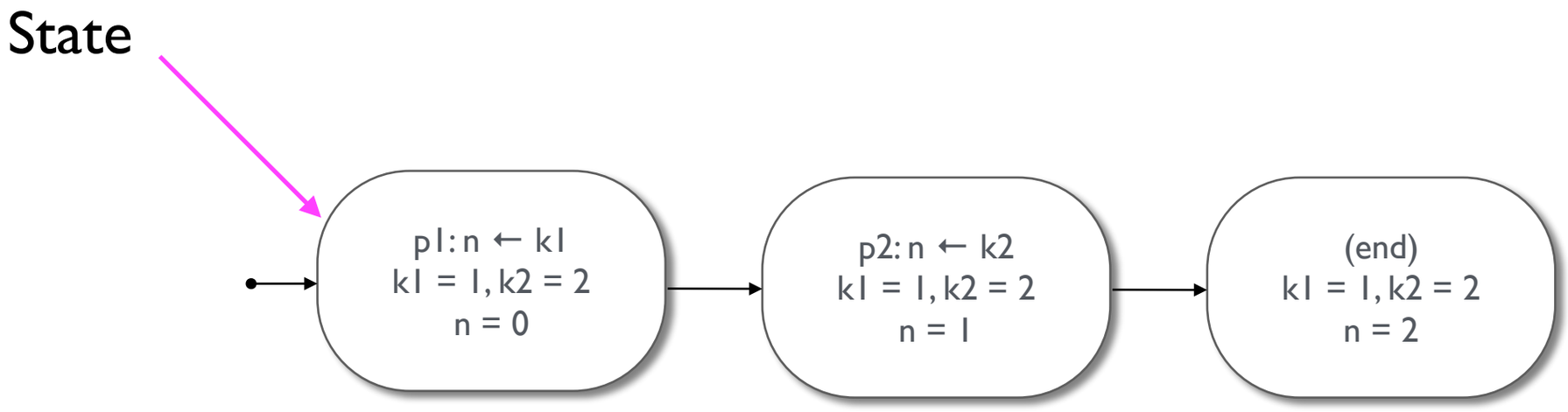
Simple Sequential Program

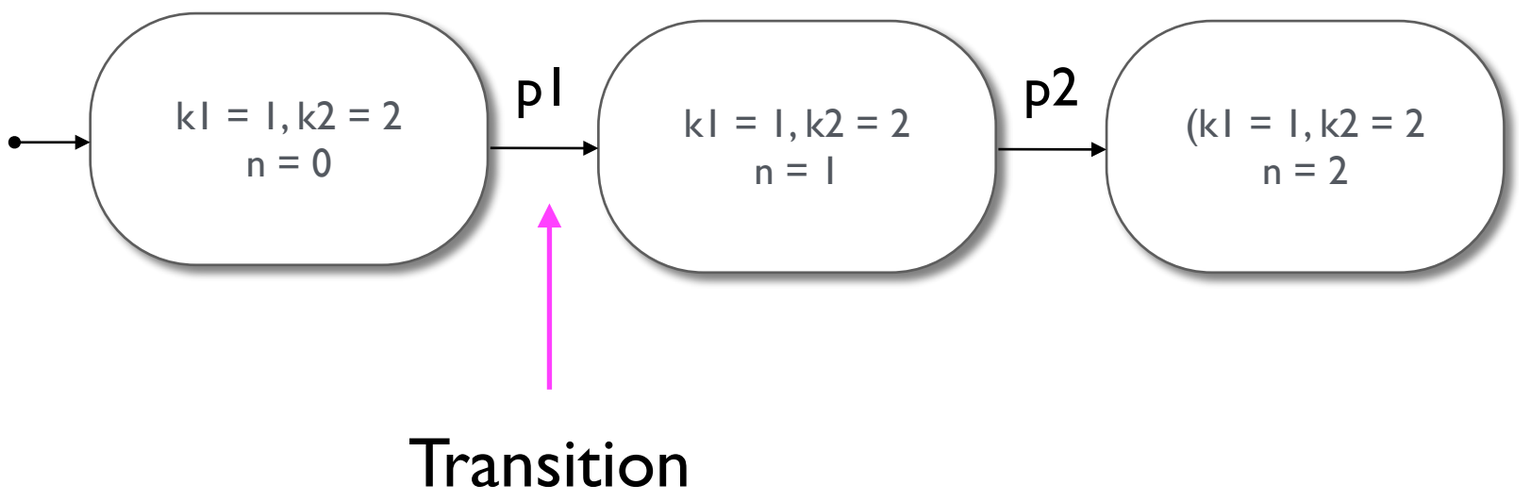
Trivial Sequential Program	
integer n ←0	
	integer k1 ← 1
	integer k2 ← 2
p1:	n ← k1
p2:	n ← k2



First line gives next atomic action to be executed

Transition Diagrams



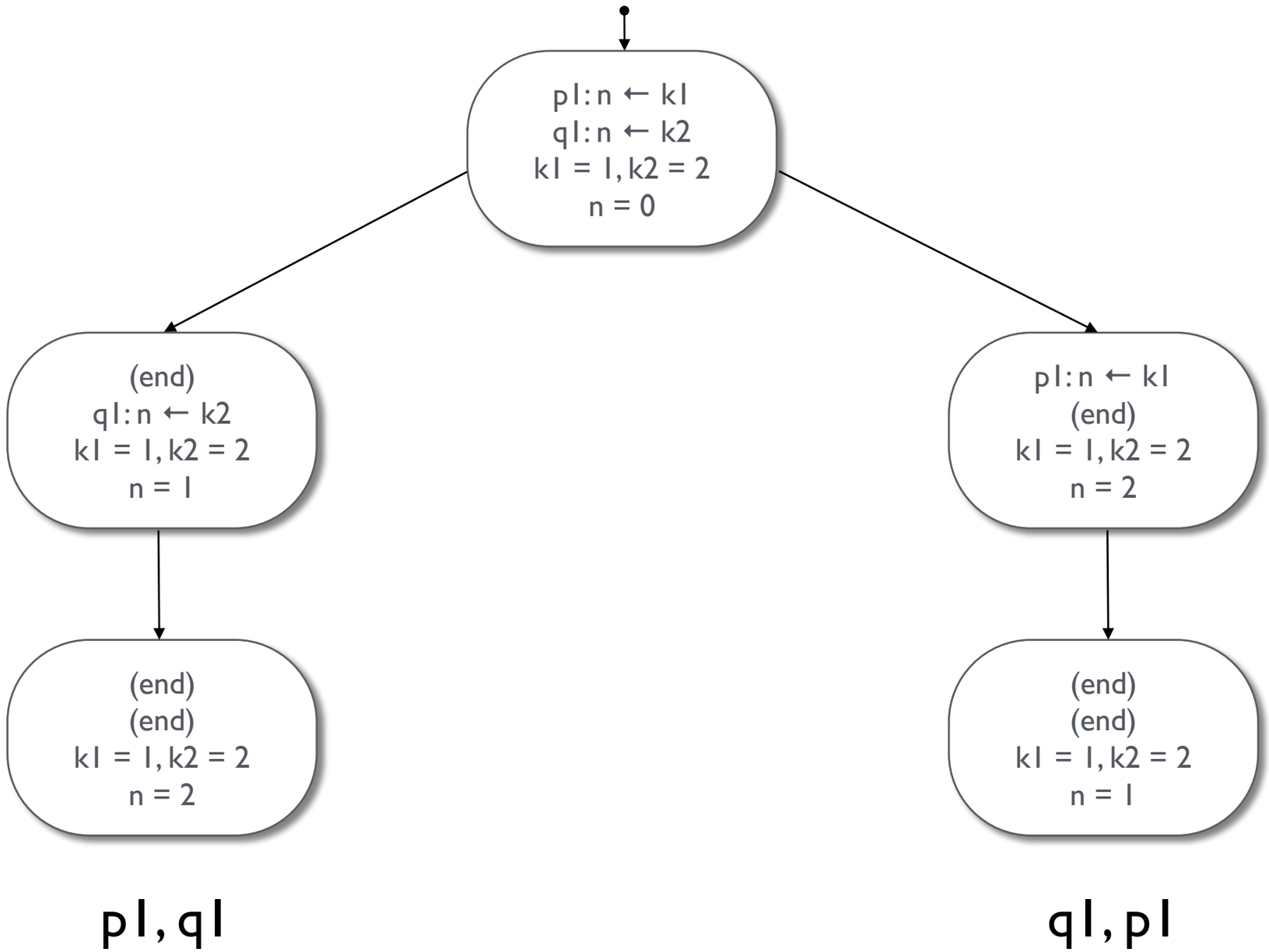


Simple Concurrent Program (1)

Trivial concurrent program: processes **p** and **q** each do one action that updates global n with the value of their local variable.

Trivial Concurrent Program	
integer n ←0 (Globals)	
p	**q**
integer k1 ← 1 (Locals)	integer k2 ← 2
p1: n ← k1 (Atomic Statements)	q1: n ← k2

Simple Concurrent Program (2)



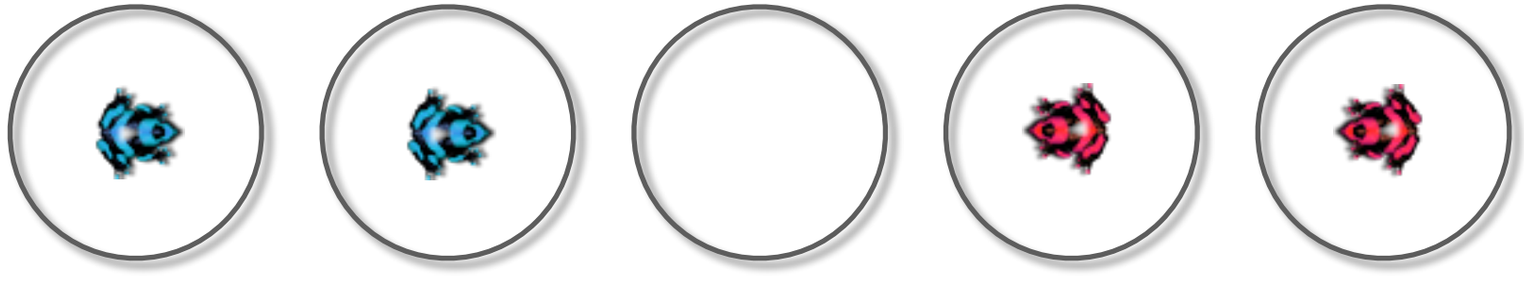
State Diagrams and Scenarios

- We could describe all possible ways a program can execute with a state diagram.
  - There is a transition between s_1 and s_2 ("s_1:s_2") if executing a statement in s_1 changes the state to s_2.
  - A state diagram is generated inductively from the starting state.
    - If $\exists$ s_1 and a transition s_1:s_2, then $\exists$ s_2 and a directed arc from s_1:s_2
- Two states are identical if they have the same variable values and the same directed arcs leaving them.
- A *scenario* is one path through the state diagram.

Example — Jumping Frogs

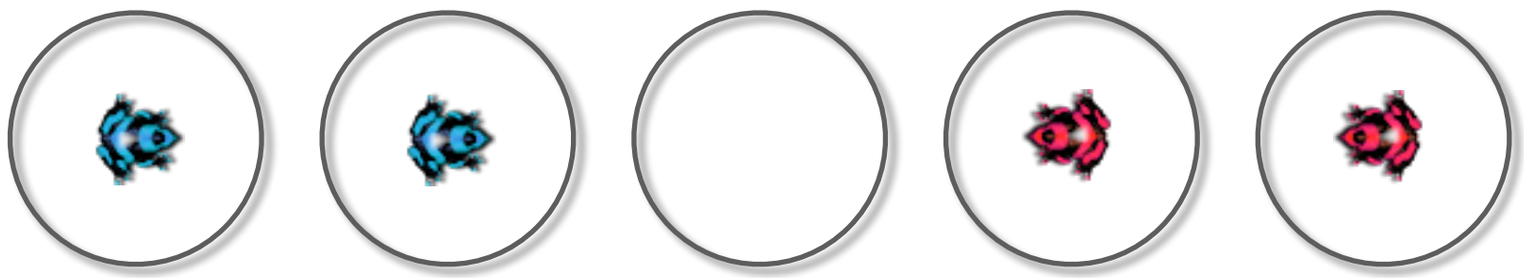
- A frog can move to an adjacent stone if it's vacant.
- A frog can hop over an adjacent stone to the next one if that one is vacant.
- No other moves are possible.

Can we interchange the positions of the frogs?



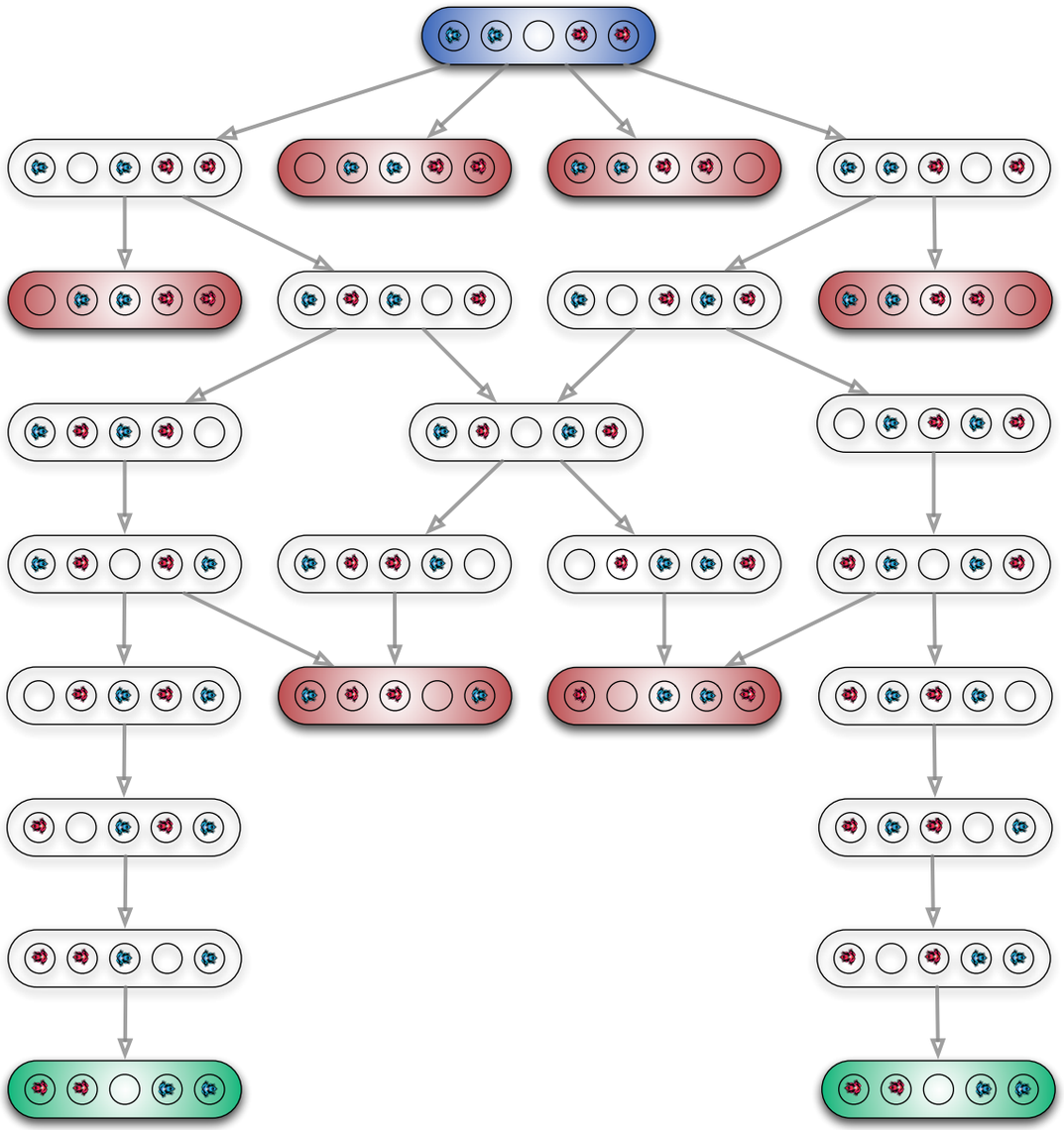
to

If the frogs can only move “forwards”, can we:



move from above to below?

Solution Graph



- So, we have a *finite state-transition diagram* of a *finite state machine (FSM)* as a *complete description* of the behaviour of the four frogs, operating concurrently, no matter what they do according to the rules.
- By examining the FSM, we can state properties as definitely holding, i.e. we can prove properties of the system being modelled.

Solution Graph

- The solution graph makes it clear that this concurrent system—of four frogs that share certain resources—can experience *deadlock*.

- *Deadlock* occurs when the system arrives in a state from which it can not make any transitions (and which is not a desired end-state.)

- *Livelock* (not possible in this system) is when the system can traverse a sequence of states indefinitely without making progress towards a desired end state.

Proof

- We can prove interesting properties by trying to construct a state diagram describing
  - each possible state of the system and
  - each possible move from one state to another
- We might use a state diagram to show that
  - A property always holds
  - A property never holds in any reachable state
  - A property sometimes holds
  - A property always holds eventually

State Diagrams for Processes

- A *state* is defined by a tuple of
  - for each process, the label of the statement available for execution.
  - for each variable, its value.
- Q: What is the maximum number of possible states in such a state diagram?

Abstraction of Concurrent Programming

- A *concurrent program* is a finite set of [sequential] *processes*.
- A process is written using a *finite set* of *atomic statements*.
- Concurrent program execution is modelled as proceeding by executing a sequence of the atomic statements obtained by *arbitrarily interleaving* the atomic statements of the processes.
- A *computation* [a.k.a. a *scenario*] is one particular execution sequence.

Atomicity

- We assume that if two operations s1 and s2 really happen at the same time, it's the same as if the two operations happened in either order.
- E.g. simultaneous writes to the same memory locations:

Sample	
integer g ← 0;	
p	q
p1: g ← 2;	q1: g ← 1

- We assume that the result will be that g will be 2 or 1 after this program, not, for example, 3.

Interleaving

- We model a scenario as an arbitrary interleaving of atomic statements, which is somewhat unrealistic.
- For a *concurrent* system, that's OK, it happens anyway.
- For a *parallel* shared memory system, it's OK so long as the previous idea of atomicity holds at the lowest level.
- For a *distributed* system, it's OK if you look at it from an individual node's POV, because either it is executing one of its own statements, or it is sending or receiving a message.
  - Thus *any* interleaving can be used, so long as a message is sent before it is received.

Level of Atomicity

- The level of atomicity can affect the correctness of a program.

Example: Atomic Assignment Statements	
integer n ← 0;	
p	q
p1: n ← n+1;	q1: n ← n+1;

Value **before** atomic statement on same row

process p	process q	n
p1: n ← n+1;	q1: n ← n+1;	0
(end)	**q1: n ← n+1;**	1
(end)	(end)	2

process p	process q	n
p1: n ← n+1;	**q1: n ← n+1;**	0
p1: n ← n+1;	(end)	1
(end)	(end)	2

Different Level of Atomicity

Example: Assignment Statements with one Global Reference	
integer n ← 0;	
p	q
integer temp	integer temp
p1: temp ← n	q1: temp ← n
p2: n ← temp + 1	q2: n ← temp + 1

Alternative Scenarios

process p	process q	n	p.temp	q.temp
p1: temp ← n	q1: temp ← n	0	?	?
p2: n ← temp+1	q1: temp ← n	0	0	?
(end)	**q1: temp ← n**	1		?
(end)	**q2: n ← temp+1**	1		1
(end)	(end)	2		

process p	process q	n	p.temp	q.temp
p1: temp ← n	q1: temp ← n	0	?	?
p2: n ← temp+1	**q1: temp ← n**	0	0	?
p2: n ← temp+1	q2: n ← temp+1	0	0	0
(end)	**q2: n ← temp+1**	1		0
(end)	(end)	1		

Atomicity & Correctness

- Thus, the level of atomicity specified affects the correctness of a program
  - We will typically assume that:
    - assignment statements and
    - condition statement evaluations
- are atomic
  - Note: this is not true for programs written in C using concurrency libraries like **`pthreads`** or similar.

Concurrent Counting Algorithm

Example: Concurrent Counting Algorithm	
integer n ← 0;	
p	**q**
integer temp	integer temp
p1: do 10 times	q1: do 10 times
p2: temp ← n	q2: temp ← n
p3: n ← temp + 1	q3: n ← temp + 1

- Increments a global variable *n* 20 times, thus *n* should be 20 after execution.
- But, the program is faulty.
  - Proof: construct a scenario where *n is 2* afterwards.
- Wouldn't it be nice to get a program to do this analysis?

pthread_create()

- Prototype:
```
int pthread_create( pthread_t *
                  , const pthread_attr_t *
                  , void *(*)(void *)
                  , void * );
```

- Call:
```
rc = pthread_create(&thread_data,NULL,ThreadCode,threadarg);
```

- Precondition: none relevant to this course... (resource availability, permissions,..)
- Postcondition: `thread_data` contains the pthread ID. `ThreadCode(threadarg)` is ready to execute.
- Invariant: none obvious but note that `ThreadCode()` may execute some of its code before `pthread_create()` returns to its caller.

pthread_exit()

- Prototype: `void pthread_exit( void * );`
- Call: `pthread_exit( NULL );`
- Precondition: Formally, none, but it's a bad idea for the calling thread to own any mutex locks.
- Postcondition: terminates the calling thread, so nothing is observed "afterwards" in the caller. In the surrounding pthread environment, any mutex locks it holds are still held afterwards. Values of variables local to the calling thread are undefined.
- Invariant: thread code that ends because the `ThreadCode` function ends normally, performs an implicit `pthread_exit(NULL);` call.

pthread_join()

- Prototype:
```
int pthread_join( pthread_t
                , void ** );
```
- Call:
```
rc = pthread_join(thread_data,NULL);
```
- Precondition: `pthread_data` refers to a existing thread; there is no other live call to `pthread_join()` with the same `pthread_data` value.
- Postcondition: The thread identified by `thread_data` has terminated.
- Invariant: `pthread_join()` does not return to the caller until the thread identified by `thread_data` has terminated; `pthread_join()` waits for termination, and does not force it.

Concurrent Counting Algorithm

Example: Concurrent Counting Algorithm	
integer n ← 0;	
p	**q**
integer temp	integer temp
p1: do 10 times	q1: do 10 times
p2: temp ← n	q2: temp ← n
p3: n ← temp + 1	q3: n ← temp + 1

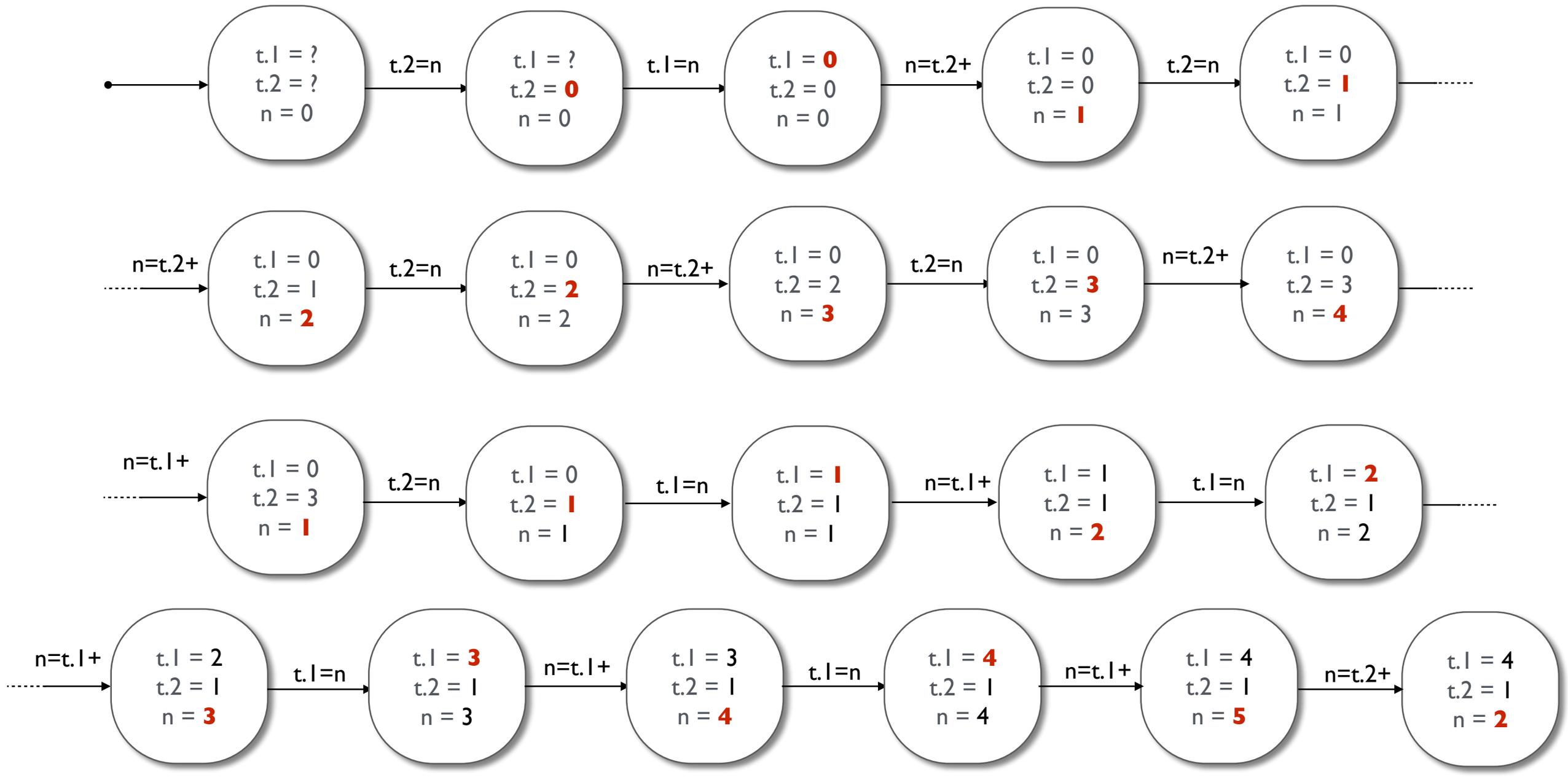
- Increments a global variable *n* 20 times, thus *n* should be 20 after execution.
- But, the program is faulty.
  - Proof: construct a scenario where *n is 2* afterwards.
- Wouldn't it be nice to get a program to do this analysis?

????

- Discovered by M. Ben-Ari during his concurrency course
  - Student puzzled him by observing a sum equal to 9
  - He modelled it and found it could be as low as 2, but no lower
- On the right, running Promela
  - with a loop of length 5 rather than 10
  - a final assertion that n>2
  - This is the counterexample resulting in not(n>2), i.e., n=2.

Process	Statement		P(1):temp	P(2):temp	n
2 P	7	temp = n			
1 P	7	temp = n	0		
2 P	8	n = (temp+1)	0	0	
2 P	7	temp = n	0	0	1
2 P	8	n = (temp+1)	0	1	1
2 P	7	temp = n	0	1	2
2 P	8	n = (temp+1)	0	2	2
2 P	7	temp = n	0	2	3
2 P	8	n = (temp+1)	0	3	3
1 P	8	n = (temp+1)	0	3	4
2 P	7	temp = n	0	3	1
1 P	7	temp = n	0	1	1
1 P	8	n = (temp+1)	1	1	1
1 P	7	temp = n	1	1	2
1 P	8	n = (temp+1)	2	1	2
1 P	7	temp = n	2	1	3
1 P	8	n = (temp+1)	3	1	3
1 P	7	temp = n	3	1	4
1 P	8	n = (temp+1)	4	1	4
2 P	8	n = (temp+1)	4	1	5
0 :init	16	_nr_pr==1	4	1	2

n is 2 scenario



Correctness, fairness

- We can use state diagram descriptions of concurrent computations to explore whether a concurrent program is *correct* according to some criterion, or whether it is *fair*.

Correctness

- (Correctness in Sequential Programs)
  - Partial Correctness: The answer is correct if the program halts
  - Total Correctness: The program does halt and the answer is correct.
- Correctness in Concurrent Programs
  - Safety Property: a property must be *always* true
  - Liveness Property: a property must be *eventually* true

Using State machines

- We can describe concurrent programs using states and state transitions.
- We can construct a state transition diagram (a *"model"*) which captures every possible scenario a concurrent program is capable of executing.
- By reasoning with the model (*"model checking"*), we can prove whether or not a particular property holds for a concurrent program.
- Much more powerful idea than merely testing a concurrent program.

Weak Fairness

- Recall a concurrent program is modelled as the interleaving of statements in any possible way in a number of sequential programs.
  - If a statement is ready for execution, then if it is *guaranteed* that it will *eventually* be executed (i.e. it will appear in a scenario), the system is said to be *weakly fair*.
- Another way of thinking about this is that in a weakly fair system, the arbitrary interleaving of statements does not include scenarios in which available and ready statements will never be executed.
- Sometimes we assume weak fairness for something to work.

Verification of concurrent programs

- Manual construction and inspection of state diagrams
  - Only suitable for the most trivial programs with few states
- Automated construction and inspection of state diagrams
  - The task of constructing the state diagram and using it to check the correctness of the concurrent program can be automated using a *model checker*
  - We still need a way to specify models and logical correctness properties
- Formal specification and verification using temporal logic

Linear Temporal Logic

- Logic that reasons about state-change over time
  - Allows us to write properties that apply to an entire scenario, i.e., sequences of states
- LTL is the simplest of a large family of temporal (and "modal") logics
  - Also often used to reason about computing systems are CTL (Computational Tree Logic) and CTL*
  - LTL is an extension of standard propositional ("digital") logic with temporal operators
  - We use the notation of the SPIN model-checker* here
    - there are more mathematical forms in the literature
    - LTL text notation is not standard - Wikipedia has a common but different variant.

* coming soon!

Propositional Logic in LTL

- LTL contains propositional logic with all the usual parts:
  - Logical constants: `true`, `false`
    - Also numeric constants: 0, 1, 2,..
  - Variables: `a`, `b`, .. `p`, `q`, `xx`, `yY`, ... - they must start with lowercase letter
    - Also boolean-valued expressions over variables and constants
  - Propositional Operators:
    - Negation: `!` Logical-and: `&&` (also `/\`) Logical-or: `||` (also `\/`)
    - Logical Implication: `->`
    - Logical Equivalence: `<->`
- We also assume that have numeric constants, and operators, and comparisons

State(s) in LTL

- A state is defined by the values of a given set of variables.
  - propositional expressions evaluate to true or false based on those values in a given ***single*** state
- LTL in general is interested in sequences (a.k.a. paths) of states.
  - In general, an LTL expression is deemed to be true of a *given starting* state in such a path:
    - if it holds true for the path from that state onwards.
- We will denote such a path as a sequence of indexed state:s
  - $S_0, S_1, S_2, ..., S_{i-1}, S_i, S_{i+1}, ...$
- A temporal property is true "at" state **S**i if it is true for the path starting with **S**i .

Linear Temporal Operators (Until)

- For most applications, all we need to do is to define one temporal operator ("until")
  - The rest of the operators can be expressed in terms of this
- LTL starts with a notion of "weak until",
  which says that (`p w-until q`) is true at starting state s_i if
  - `q` is true at **S**i, or
  - `p` is true at **S**i and (`p w-until q`) is true at **S**i+1
  - Note that there is no requirement for `q` to ever be true, but if it is never so, then `p` must always be true
- "Strong until" (`s-until`) is "weak until" with an additional requirement that `q` must become true after a finite number of steps.
- SPIN uses the notation `U` to denote strong until - it does not have the weak operator.

Linear Temporal Operators (derived)

- We can now derive two other useful operators
  - one using "weak-until", the other using "strong-until"
- Always p (`[]p`) is true if p is true in every state in the path
  - It can be defined using weak-until as `[]p = p w-until false .`
- Eventually p (`<>p`) is true if p is true at least once, somewhere along path, after a finite number of steps
  - It is defined using strong-until as `<>p = true U p`

Always

- Always p (`[]p`) is true at **S**i if p is true in every state in the path from **S**i onwards
  - It can be defined using weak-until as `[]p = p w-until false`.
- (`p w-until false`) is true at starting state Si if
  - `false` is true at **S**i, (`false` is never true anywhere, anytime!) or
  - p is true at **S**i and (`p w-until false`) is true at **S**i+1
  - Note that there is no requirement for `false` to ever be true, but if it is never so, then p must always be true
- So p must always be true!

Eventually

- Eventually q (<>q) is true at state **S**i if q is true at least once, somewhere along the path from **S**i, after a finite number of steps
  - It is defined using strong-until as `<>q = true U q`
- which says that (`true U q`) is true at starting state **S**i if
  - `q` is true at **S**i, or
  - `true` is true at **S**i (it always is!) and (`true U q`) is true at **S**i+1
  - `q` must be true after a finite number of steps along the path.
- So q must be true after a finite number of steps.

predicates without temporal operators

- We refer to a predicate without temporal operator as a "state predicate"
  - It is a predicate using constants, expressions, and the proposition operators.
- State predicate p is true "at **S**i" if it is true of **S**i .
  - There is no reference to any subsequent states.

Nested temporal operators

- The ps and qs used previously to talk about arguments to operators like "until", "always", "eventually" can themselves be built using such operators: [][]p) <>([]P -> <>Q)
- We can define a systematic set of rules that can always determine if a given path of states is satisfied by a given temporal logic predicate.
  - These rules apply recursively
  - They are easy to automate

Common LTL Predicates

LTL	Reads as...	Property
`[]p`	always p	invariance
`<>p`	eventually p	guarantee
`p -> <>q`	p implies eventually q	response
`p -> q U r`	p implies q until r	precedence
`[]<>p`	always eventually p	recurrence (progress)
`<>[]p`	eventually always p	stability (non-progress)
`<>p -> <>q`	eventually p implies eventually q	correlation

("The Spin Model-Checker", p137)

Common LTL Predicates

LTL	Reads as...	Property
`[]p`	always p	invariance
`<>p`	eventually p	guarantee
`p -> <>q`	p implies eventually q	response
`p -> q U r`	p implies q until r	precedence
`[]<>p`	always eventually p	recurrence (progress)
`<>[]p`	eventually always p	stability (non-progress)
`<>p -> <>q`	eventually p implies eventually q	correlation

("The Spin Model-Checker", p137)

Model-Checking

- Model checking is an automated technique that, given a finite-state model of a system and a logical property, systematically checks whether this property holds for that model, given an initial state.
- Due to initial work by Clarke and Emerson (1980, 1981, 1986).
- Model checking tools automatically check whether M ⊨ φ holds, where M is a (finite-state) model of a system and property φ is stated in some formal notation.
- SPIN is one of the most powerful model checkers
  - It uses Linear Temporal Logic
  - Its model M is the set of all possible paths from the starting state.

Promela

- Promela [*Pro*tocol/*Pro*cess *M*eta *La*nguage] is a modelling language, not a programming language.
- A Promela model consist of:
  - type declarations
  - channel declarations
  - global variable declarations
  - process declarations
  - [init process]

`mtype`, constants, typedefs (records)

- `simple vars`
- `structured vars`

behaviour of the processes: local variables + statements

initialises variables and starts processes

- A Promela model corresponds to a *finite* transition system, so:
  - no unbounded data / channels / processes / process creation

SPIN

- SPIN (Simple ProMeLa INterpreter) is a model-checking based verification tool for concurrent systems, e.g. concurrent programs.
- SPIN can also be used to ‘run’ the model, almost as if it was a program, to informally examine it.

Process (1)

- A process is defined by a **`proctype`** definition
- It executes concurrently with all other processes, independently of speed or behaviour
- A process communicates with other processes:
  - using global variables
  - using channels
- There may be several processes of the same type.
- Each process has its own local state:
  - the process counter (location within the proctype)
  - the contents of the local variables

Process (2)

- A process type (`proctype`) consists of
  - a name
  - a list of formal parameters
  - local variable declarations
  - the body

```
proctype Sender(byte in; byte out) {
    bit sndB;
    do
    :: out==10 ->
            in=4;
            if
            :: sndB == 1 -> sndB = 1-sndB
            :: else -> skip
            fi
    od
}
```

Hmmm, funny looking code up there ....

Process (3)

- Processes are created using the **`run`** statement (which returns the process id).
- Processes can be created at any point in the execution.
- Processes start executing after the **`run`** statement.
- Processes can also be created and start running by adding **`active`** in front of the **`proctype`** declaration.

Yes, that is an array of **`P`** ! **`P[0]`** and **`P[1]`**.

```
active [2] proctype P() {
    byte    i = 1;
    byte    temp;
    do
        :: ( i > TIMES ) -> break
        :: i > 10 ->
        temp = n;
        n = temp + 1;
        i++
    od;
    finished++;  /* Process terminates */
}
```

More funny looking code ....

SPIN Hello World Example

SPIN = *S*imple *P*romela *In*terpreter

```
/* A "Hello World" Promela model for SPIN. */
active proctype Hello() {
    printf("Hello process, my pid is: %d\n", _pid);
}
init {
    int lastpid;
    printf("init process, my pid is: %d\n", _pid);
    lastpid = run Hello();
    printf("last pid was: %d\n", lastpid);
    lastpid = run Hello();
    printf("last pid was: %d\n", lastpid);
}
```

```
$ spin -n2 hello.pr
          init process, my pid is: 1
          last pid was: 2
      Hello process, my pid is: 0
              Hello process, my pid is: 2
          last pid was: 3
                  Hello process, my pid is: 3
4 processes created
```

Promela Types

- Basic types
  - **`bit`** e.g. turn=1; range: [0..1]
  - **`bool`** e.g. flag; [0..1] or **`true`**, **`false`**
  - **`byte`** e.g. counter; [0..255]
  - **`short`** e.g. s; [-2^{15}.. $2^{15}-1$]
- Default initial value of basic variables (local and global) is 0.
- Most arithmetic, relational, and logical operators of C are supported, including bitshift operators.

Promela Types (2)

- Arrays
  - Zero-based indexing (one-dimensional only !)
- Records (“structs” in C/)
  - typedefs
- Mtypes
  - Promela supports one enumeration type called “**mtype**”.
  - It’s simply a set of names to be treated as distinct values

```
typedef Record {
  short f1;
  byte f2;
}
```

```
mtype = {name1, name2, … , nameN }
```

Special Variables

- Promela has special variables that make some internal properties visible.
  - These variables have names that start with an underscore ('_').
- Examples:
  - **`_pid`** return the process id of the current process
  - **`_nr_pr`** returns the number of currently active processes.

Statements

- A statement is either
  - *executable*: the statement can be executed immediately.
  - *blocked*: the statement cannot be executed.
- An assignment is always executable.
- An expression is also a statement; it is executable if it evaluates to non-zero. E.g.
  - 2 < 3 always executable
  - x < 27 only executable if value of x is smaller 27
  - 3 + x executable if x is not equal to –3
  - "Executing" an expression has no effect on program variable state other than updating the process counter.

Statements (2)

- The skip statement is always executable.
  - it "does nothing", only changes the process counter
- A run statement is only executable if a new process can be created (remember: the number of processes is bounded).
- A printf statement is always executable (but ignored, i.e. not considered, during verification).

```
int x;
proctype Aap()
{
  int y=1;
  skip;
  run Noot();
  x=2;
  x>2 && y==1;
  skip;
}
```

Let's demo the above!

Statements (3) — assert

- Format: **`assert(`**<expr>**`);`**
- The assert statement is always executable.
- If <expr> evaluates to zero, SPIN will exit with an error, as the *<expr> has been violated.*
- Often used within Promela models, to check whether certain properties are valid in a state.

sumofhellos.pml

- We can use Promela to model the **`sumofhellos.c`** C code used for Practical I
- We create a file called **`sumofhellos.pml`**
  - We use expression **`(_nr_pr == 1)`** to wait until all the **`PrintHello`** threads are done (i.e., until only the **`init`** process is still running).
- We can run a random simulation using SPIN on the command line:
  - **`$> spin sumofhellos.pml`**
- We see output similar to that of the C program.

Statements (4)

- All the statements mentioned so far (a.k.a. Basic statements) are ***atomic***:
  - when executable, once they execute, they do so atomically.
- Promela also has composite statements that compose other statements in some way
  - Sequential composition
  - Conditionals
  - Loops
- Composite statements are not atomic

Sequencing Statements

```
stat1 ; stat2 ; stat3 ; … ; statN
```

- We sequence statements simply by listing them, ***separated*** by semi-colons (‘*;*’).
- This is unlike in C, where ‘*;*’ is a statement *terminator*.
- Promela is liberal about semi-colons, so it will accept a last semicolon

```
stat1 ; stat2 ; stat3 ; … ; statN ;
```

or many !

```
stat1 ; stat2 ; stat3 ; … ; statN ;;;
```

If Statement

```
if
:: (n % 2 != 0) -> n=1
:: (n >= 0)     -> n=n-2
:: (n % 3 == 0) -> n=3
:: else         -> skip
fi
```

It's that weird code again!

- Each `::` introduces an alternative started by convention with a guard statement (expression)
  - if any guards are executable, then one is non-deterministically chosen.
  - Once an alternative has run, the conditional itself has terminated.
  - The optional else becomes executable if none of the other guards are executable.
- If no guard is executable, the if statement blocks, until least one becomes executable.
- The notation '->' is simply another way of writing ';'

Do Statement

```
do
 :: choice1 -> stat1.1; stat1.2; stat1.3; …
 :: choice2 -> stat2.1; stat2.2; stat2.3; …
 :: …
                :: choicej -> break;
 :: choicen -> statn.1; statn.2; statn.3; …
od;
```

A cunningly disguised while-loop!

- Similar to the If statement, but it repeats the choice at the end.
- Use the break statement to move on to the next statement after **`od`**.
- The strange notation derives from so-called Guarded Command Language (GCL)
  - Developed by Edsker Dijkstra to reason about program correctness
  - Its form is designed to emphasise any non-determinism present in a system.
  - The boolean expressions **`choice1, choice2`**,... are called the "guards".

Atomic Statement

```
atomic { stat1; stat2; ... statn }
```

- An `atomic{}` statement can be used to group statements into an atomic sequence;
- all statements are executed in a single sequence (no interleaving with statements of other processes), though each step is taken.
- The statement is executable if `stat1` is executable
- If a `stat i` (with i>1) is blocked, the "atomicity" is temporarily lost and other processes may do a step.

Sumofhellos made atomic

- We can use the **`atomic`** construct to easily produce a version of sumofhellos that works properly (**`atm_sumofhellos.pml`**).
  - This is the advantage of a modelling language over a program language:
    We can say "make it so!" (to some extent, at least).
- We can run simulations of these, but how do we know we have fixed things?
- The real strength of SPIN is it can build the state diagram and check every possible path through it, so determining the outcome of every possible interleaving
  - **`$> spin -run atm_sumofhellos.pml`**

`#define` and `inline`

- SPIN uses the C pre-processor (CPP) to process Promela files
  - So all the CPP facilities are available, such as **`#define`**, **`#if`**, **`#ifdef`**, etc.
- There is also something similar called **`inline`**.
  - It has to be defined at the top level

```
inline name(arg1, …, argN){
   stat1; stat2; ... statn
}
```

  - It can only be called where a statement can occur

```
name(val1, …, valN);
```

  - An **`inline`** can contain calls to ***other*** **`inline`**s (but cannot be recursive).

inline behaviour

```
inline name(arg1, …, argN){
   stat1; stat2; ... statn
}
```

- Important: **`inline`** describes ***textual substitution***.
  - Just like the way **`#define`** operates.
- The inline code does not represent a function/procedure that can be called.
  - A call **`name(val1, …, valN);`** results in the texts "**`val1`**", .. "**`valN`**" being substituted for the occurrences of "**`arg1`**", .. "**`argN`**", wherever they appear.
- In particular, if a statement like **`stat1`** (say) declares a variable, then that variable has ***global*** scope.

`#define` vs. `inline`

- What is the difference?
- If an error occurs in code produced by a macro defined using **`#define`**,
  - the error is reported at the point of use in the expanded macro text
- If an error occurs in code produced by a macro defined using **`inline`**,
  - the error is reported at the relevant line in the **`inline`** definition itself
    - Generally much more useful.

Modelling Mutexes in Promela

- We can model mutexes in Promela as a variable whose state is locked or unlocked with the check for being not locked done atomically with locking it. We also record the **`_pid`** of the process with the lock.

```
mtype = { unlocked, locked }
mtype mutex = unlocked ;
int mid = 0;
```

- We want to check a mutex is unlocked and then lock it atomically:

```
atomic{ mutex==unlocked -> mutex = locked; mid = _pid } ;
```

- To unlock, we should have the mutex lock, so we need to check our **`_pid`**

```
atomic {
  assert(mid==_pid);
  mutex = unlocked;
  mid = 0;
}
```

Sumofhellos using mutexes

- We can use the inline construct to easily produce a version of sumofhellos that uses mutexes. (**`mtx_sumofhellos.pml`**).
  - This is the advantage of a modelling language over a program language:
    We can say "make it so!" (to some extent, at least).
- As before, we can run simulations of this.
- Again, we can build the state diagram and check every possible path through it, so determining the outcome of every possible interleaving
  - **`$> spin -run mtx_sumofhellos.pml`**

Concurrent Counting Algorithm (Revisited)

Example: Concurrent Counting Algorithm	
integer n ← 0;	
p	**q**
integer temp	integer temp
p1: do 10 times	q1: do 10 times
p2: temp ← n	q2: temp ← n
p3: n ← temp + 1	q3: n ← temp + 1

- Increments a global variable *n* 20 times, thus *n* should be 20 after execution.
- But, the program is faulty.
  - Proof: construct a scenario where *n is 2* afterwards.
- Wouldn't it be nice to get a program to do this analysis?

????

- Discovered by M. Ben-Ari during his concurrency course
  - Student puzzled him by observing a sum equal to 9
  - He modelled it and found it could be as low as 2, but no lower
- On the right, running Promela
  - with a loop of length 5 rather than 10
  - a final assertion that n>2
  - This is the counterexample resulting in not(n>2), i.e., n=2.

Process	Statement		P(1):temp	P(2):temp	n
2 P	7	temp = n			
1 P	7	temp = n	0		
2 P	8	n = (temp+1)	0	0	
2 P	7	temp = n	0	0	1
2 P	8	n = (temp+1)	0	1	1
2 P	7	temp = n	0	1	2
2 P	8	n = (temp+1)	0	2	2
2 P	7	temp = n	0	2	3
2 P	8	n = (temp+1)	0	3	3
1 P	8	n = (temp+1)	0	3	4
2 P	7	temp = n	0	3	1
1 P	7	temp = n	0	1	1
1 P	8	n = (temp+1)	1	1	1
1 P	7	temp = n	1	1	2
1 P	8	n = (temp+1)	2	1	2
1 P	7	temp = n	2	1	3
1 P	8	n = (temp+1)	3	1	3
1 P	7	temp = n	3	1	4
1 P	8	n = (temp+1)	4	1	4
2 P	8	n = (temp+1)	4	1	5
0 :init	16	_nr_pr==1	4	1	2

2 ways to run SPIN

- SPIN can be run in one of two modes: *Simulation* and *Verification*
- *Simulation*: SPIN performs ***one*** possible run of the system, making its own choices
  - such runs are often referred to as "*Scenarios*"
  - usually choices are random, and we can use command-line options to control the randomness
  - SPIN can also do a guided simulation, taking input from a so-called "trail" file (see below)
- *Verification*: SPIN systematically searches over ***all*** possible runs of the system
  - Checking for the truth of desirable properties
  - If a check fails, it outputs a *Counter-example*.
- *Counter-example:* a run of the system that leads to a property failure
  - output to a "trail" file

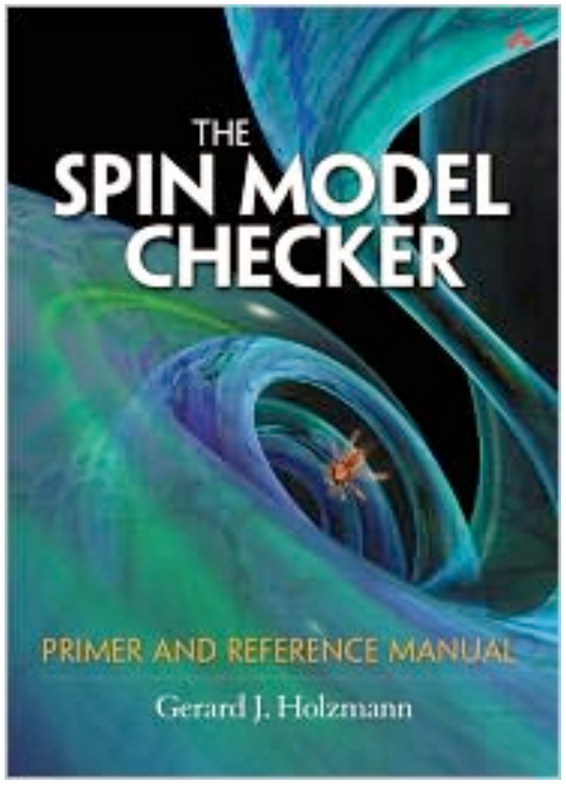
Sumofhellos mis-using mutexes

- Let's produce a version of sumofhellos that uses mutexes in an incorrect manner. (bad**`_sumofhellos.pml`**).
  - This is the advantage of a modelling language over a program language: We can say "make it so!" (to some extent, at least).
- As before, we can run simulations of this, and we observe failure.
- Again, we can build the state diagram and check every possible path through it, so determining the outcome of every possible interleaving
  - **`$> spin -run bad_sumofhellos.pml`**
- In this case we not only see an error indication, but a "trail" file has been created, which we can run with
  - **`$> spin -p -k bad_sumofhellos.pml.trail bad_sumofhellos.pml`**

Sumofhellos mis-using mutexes

- Let's produce a version of sumofhellos that uses mutexes in an incorrect manner. (bad**`_sumofhellos.pml`**).
  - This is the advantage of a modelling language over a program language: We can say "make it so!" (to some extent, at least).
- As before, we can run simulations of this, and we observe failure.
- Again, we can build the state diagram and check every possible path through it, so determining the outcome of every possible interleaving
  - **`$> spin -run bad_sumofhellos.pml`**
- In this case we not only see an error indication, but a "trail" file has been created, which we can run with
  - **`$> spin -p -k bad_sumofhellos.pml.trail bad_sumofhellos.pml`**

Book Reference



The SPIN Model Checker: Primer and Reference Manual
By Gerald Holtzmann
Addison-Wesley, May 2011 ISBN 0321773713
$65 [Paperback]

The spin model checker : primer and reference manual /
Gerald J. Holzmann
Holzmann, Gerard J.

Printed Book | 2004

Available at Hamilton Lower (S-LEN 620 P496) plus 3 more see all

Click and Collect

Additional actions:

Modelling and validating 1553B p
model checker
Krishnan, Ranjani;Lalithambika, V R

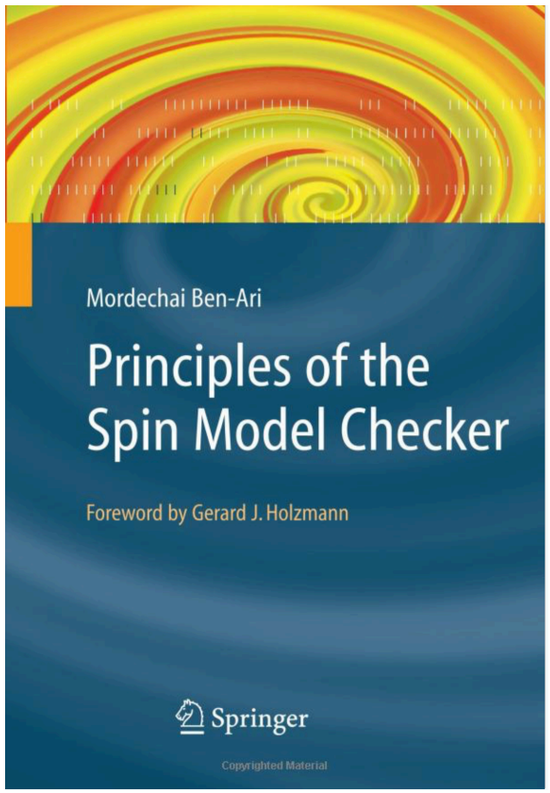
Conference | 2018 10th International C
Systems & Networks (COMSNETS) Commu
(COMSNETS), 2018 10th International Conf

TCD affiliated users click here for access

Items only show available close

Location (click for map)	Shelfmark	Status
Hamilton Lower	S-LEN 620 P496	IN
Hamilton Lower	S-LEN 620 P496;2	IN
Hamilton Lower	S-LEN 620 P496;3	IN
Hamilton Upper	620 P496;1	IN

Book Reference



Principles of the Spin Model Checker 2nd Edition
By Mordechai Ben-Ari
Springer, 2008 ISBN 978-1-84628-769-5
$63 [Paperback]

Principles of the Spin model checker / **Mordechai Ben-Ari**
Ben-Ari, M., 1948-

Printed Book | 2008

Not Available at Hamilton Lower (DUE 16-04-21) see all

Click and Collect

Additional actions:

→ Publisher description Click he
→ Table of contents only Click h

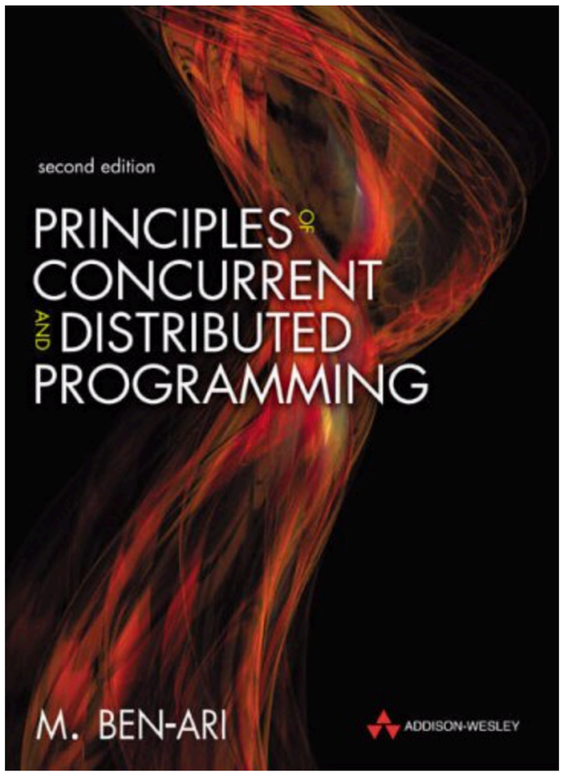
Blockchain based permis
control in Internet of Thi
Ali, Gauhar;Ahmad, Naveed;Cao,

Items

close

Location (click for map)	Shelfmark	Status
Hamilton Lower	S-LEN 500.164 P83;1	DUE 16-04-21
Hamilton Upper	500.164 P83	DUE 16-04-21

Book Reference



Principles of Concurrent and Distributed Programming
2nd Edition
By Mordechai Ben-Ari
Addison-Wesley, 2006 ISBN 978-0-321-31283-9
$120 [Paperback]

Principles of concurrent and distributed programming / **M. Ben-Ari**
Ben-Ari, M., 1948-

Printed Book | 2006

Available at Hamilton Lower (S-LEN 500.164 N093*1) plus 4+ more see all

Click and Collect

Additional actions:

Principles of concurrent and distribute
M. Ben-Ari
Ben-Ari, M., 1948-

Printed Book | 1990

Available at Santry Stacks (PL-157-209) see all

Items only show available cl

Location (click for map)	Shelfmark	Stat
Hamilton Lower	S-LEN 500.164 N093*1	IN
Hamilton Lower	S-LEN 500.164 N093*1;1	IN
Hamilton Lower	S-LEN 500.164 N093*1;2	IN
Hamilton Lower	S-LEN 500.164 N093*1;3	IN

Mutual Exclusion (without pthreads!)

- Early challenge in concurrency was to provide mutual exclusion
  - assuming that the only atomic operations were read and write
- First algorithm, for two processes only, described in 1965 by Edsger Dijkstra
  - He attributed it to T.J Dekker in 1962.
- It does not assume that two processes alternate access
    - It allow each process to return to the critical region as many times as it wants before the other one gets access.

Dekker's Algorithm (I)

```
bool turn;
bool flag[2];
byte count;

active [2] proctype mutex()
{    pid i, j;

    i = _pid;
    j = 1 - _pid;

again:

    < ACQUIRE "LOCK">

    count++;
    assert(count == 1); /* critical section */
    count--;

    <RELEASE "LOCK">

    goto again
}
```

- Overview:
  - variable **`count`** is the global shared variable
  - variable **`i`** refers to thread's own id, while **`j`** is that of the other thread.
- Body is an infinite loop
  - defined using label **`again:`**
  - and a **`goto`** statement !

Dekker's Algorithm (II)

```
flag[i] = true;
do
:: flag[j] ->
    if
    :: turn == j ->
        flag[i] = false;
        !(turn == j);
        flag[i] = true
    :: else
    fi
:: else ->
    break
od;
```

- Acquiring the lock:
  - set own **`flag`** to say "going in!"
  - while other **`flag`** is true:
    - if other's **`turn`**:
      turn own **`flag`** off, wait for turn, turn **`flag`** on again
- When we exit, the other's **`flag`** is off and it is our **`turn`**.

("The Spin Model-Checker", p20)

Dekker's Algorithm (III)

```
turn = j;
flag[i] = false;
```

- Releasing the lock:
  - It is their **`turn`**
  - Our **`flag`** is now false.

("The Spin Model-Checker", p20)

Checking Dekker

- We can run it: `spin -u1000 dekker.pml`
- We can verify it `spin -a dekker.pml ; cc -o pan pan.c ; ./pan`
- ????

- SPIN takes a Promela model and compiles it into a C program that builds and analyses that model.
- The executable is usually called "pan"
  - short for "Protocol Analyser" - the first application area for Promela/SPIN
- splitting **`spin -run model.pml`** into **`spin -a model.pml`** and then using **`pan`** allows a lot of flexibility in the kinds of analyses that can be done.

Improvement (Doran&Thomas 1980)

```
flag[i] = true;
if
:: flag[j] ->
    if
    :: turn == j ->
        flag[i] = false;
        !(turn == j);
        flag[i] = true
    :: else
    fi;
    (!flag[j]);
:: else ->
fi;
```

- Is top-level loop necessary?
  - Hard to say without careful analysis
- Acquiring the lock:
  - set own **`flag`** to say "going in!"
  - if other **`flag`** is true:
    - if other's **`turn`**:
      turn own **`flag`** off, wait for **`turn`**, turn **`flag`** on again
    - then, wait for their **`flag`** to turn off.
      - Not in SPIN/Promela book !
- When we exit, the other's **`flag`** is off and it is our **`turn`**.

("The Spin Model-Checker", p22, plus note in Errata)

```
byte count;
byte x, y, z;
active [2] proctype user()
{ byte me = _pid + 1; /* me is 1 or 2 */
L1: x = me;
L2: if
  :: (y != 0 && y != me) -> goto L1
  :: (y == 0 || y == me)
  fi;
L3: z = me;
L4: if
  :: (x != me)  -> goto L1
  :: (x == me)
  fi;
L5: y = me;
L6: if
  :: (z != me) -> goto L1
  :: (z == me)
  fi;

L7: progress:/* success: enter critical section */
  count++;
  assert(count == 1);
  count--;
  goto L1
}
```

Something Broken

- A major computer manufacturer recommended this....
  - SPIN very quickly finds a counter-example!

```
spin -run mutex_flaw.pml
```

Shrinking SPIN's counter-example

- Failure is noted and a .trail file created
  - the "depth" is how many steps were taken to failure
  - We can "run" the trail: **`spin -p -t mutex_flaw.pml`**

- Two ways to find a shorter counter-example
  - Keep track of depth and iteratively search for minimal depth
    `cc -DREACH -o pan pan.c ; ./pan -i -mDD` where DD is known depth.
  - Use breadth-first search instead (default is depth-first)
    `cc -DBFS -o pan pan.c ; ./pan`

The Closed World Assumption

- A Model Checker seeks to establish the truth of something by trying exhaustively and failing to demonstrate the ***negation*** of it.
  - e.g. to show that mutex holds, it tries to demonstrate (by counterexample) that it doesn't hold. If it fails to do so, it is taken as proof that the mutex holds.
- This only corresponds to what we normally understand to be "proof" if it is assumed that the model checker knows everything (i.e. can prove everything provable) about the system -- it's called the "Closed World Assumption."
- A property deemed true, or discovered to be false, by a model-checker, may not be so in the real world.
  - This typically arises because a model only covers some aspects of a systems behaviour, and other aspects, not modelled, may also have an impact on behaviour.

Promela Verification Constructs

- So far, we have seen that SPIN does some standard verification checks by default
  - e.g. Deadlock Freedom
- There are a number of other common cases that can be handled without using Linear Temporal Logic (LTL)
  - Basic Assertions
  - Special Labels:
    - End-States
    - Progress-States

Promela – *Basic Assertion*

- Format: assert(<expression>)
  - where `<expression>` must evaluate to `true` or
    - simulation will be aborted or
    - verification will fail

      Note: an `assert` is evaluated during both simulation and verification.
- Already seen its use in **`sumofhellos.pml`**, for example

Promela Labels

- Promela allows labels to be attached to statements
  - (e.g. see **`dekker.pml`** or **`doran.pml`**)
- By using labels with a specific ***prefix***, we can trigger certain kinds of correctness checks.

Promela – *End-State Label*

- Promela always verifies that no deadlock occurs.
- It assumes that the only valid end-states for a system are where:
  - each process is the end of its code.
- If it can show an end state where a process is not at the end of its code, it considers that an error (Deadlock).
- Sometimes, it is legitimate for a process not to end up at the end of its code. You can label such states **`end`**... to indicate to Promela that they are valid end points.

  `end`, **`end_one`**, **`end00`** – anything starting with **`end`**.

end...: label example

- An example of a valid-end state, that is not the end of a program, is where a program that is not meant to terminate (i.e. a web-server) is waiting for client requests to arrive.
- Consider an example of two servers talking to a client that accesses them both and terminates [SMC-programs/ch04/end.pml]
  - When the client terminates, both servers are left waiting on a request
  - SPIN reports an error - but this is erroneous

```
1  /* Copyright 2007 by Moti Ben-Ari
2  under the GNU GPL; see readme.txt */
3
4  byte request = 0;
5
6  active proctype Server1() {
7    do
8    :: request == 1 ->
9            printf("Service 1\n");
10           request = 0;
11   od
12 }
13
14 active proctype Server2() {
15   do
16   :: request == 2 ->
17           printf("Service 2\n");
18           request = 0;
19   od
20 }
21
22 active proctype Client() {
23   request = 1;
24   request == 0;
25   request = 2;
26   request == 0;
27 }
```

`end...:` label example

- We fix this by labelling the (start of) the **`do .. od`** statement in each server.
- SPIN is now happy

```
1  /* Copyright 2007 by Moti Ben-Ari
2  under the GNU GPL; see readme.txt */
3
4  byte request = 0;
5
6  active proctype Server1() {
7  end1: do
8        :: request == 1 ->
9                printf("Service 1\n");
10               request = 0;
11       od
12 }
13
14 active proctype Server2() {
15 end2: do
16       :: request == 2 ->
17               printf("Service 2\n");
18               request = 0;
19       od
20 }
21
22 active proctype Client() {
23   request = 1;
24   request == 0;
25   request = 2;
26   request == 0;
27 }
```

Promela – *Progress-State Label*

- A system can have loops – cycles of sequences of states passed through infinitely often.
  - The question is, are such loops desirable or not;
    - if progress is made each time, then they are desirable loops -- progress cycles;
    - otherwise, they are undesirable *non-progress cycles*, where the system is doing something but not progressing.
- A given loop might never get executed
  - This can happen if there is always something else that can run elsewhere, and the "scheduler" is being "unkind" to this loop
  - If the loop contains a state that should be happening repeatedly, then this is a case of ***starvation***.
- You can label such states **`progress`**... to indicate to Promela that they should occur in any (infinite) loop.

  `progress`, **`progress_one`**, **`progress00`** – anything starting with **`progress`**.

`progress...:` label example

- Here we another implementation of mutual exclusion
  - It is deadlock-free and only one thread is in critical region at any time.
  - A standard SPIN verification run shows no problems
- However, can one thread run at the expense of the other?

```
1  /* Copyright 2007 by Moti Ben-Ari
2  under the GNU GPL; see readme.txt */
3
4  bool wantP = false, wantQ = false;
5
6  active proctype P() {
7    do
8    :: wantP = true;
9       do
10      :: wantQ -> wantP = false; wantP = true
11      :: else -> break
12      od;
13      // critical section
14      wantP = false
15   od
16 }
17
18 active proctype Q() {
19   do
20   :: wantQ = true;
21      do
22      :: wantP -> wantQ = false; wantQ = true
23      :: else -> break
24      od;
25      // critical section
26      wantQ = false
27   od
28 }
```

`progress...:` label example

- We can use a progress label
  - We label the critical section in **`P()`**
- We need a special verifier:
  - Optimised for "non-progress" (NP) cycles

    `cc -DNP -o pan pan.c`
- We need to ask it to find such cycles

  `./pan -l`

```
1  /* Copyright 2007 by Moti Ben-Ari
2  under the GNU GPL; see readme.txt */
3
4  bool wantP = false, wantQ = false;
5
6  active proctype P() {
7    do
8    :: wantP = true;
9       do
10      :: wantQ -> wantP = false; wantP = true
11      :: else -> break
12      od;
13      progressP: // critical section
14      wantP = false
15   od
16 }
17
18 active proctype Q() {
19   do
20   :: wantQ = true;
21      do
22      :: wantP -> wantQ = false; wantQ = true
23      :: else -> break
24      od;
25      // critical section
26      wantQ = false
27   od
28 }
```

Showing infinite cycles

- SPIN has a way of showing infinite cycles

```
:- spin -t -p SMC-5-1.pml
starting claim 2
spin: couldn't find claim 2 (ignored)
  2:    proc  1 (Q:1) SMC-5-1.pml:21 (state 1)  [wantQ = 1]
  <<<<<START OF CYCLE>>>>>
  4:    proc  1 (Q:1) SMC-5-1.pml:24 (state 5)  [else]
  6:    proc  1 (Q:1) SMC-5-1.pml:27 (state 10) [wantQ = 0]
  8:    proc  1 (Q:1) SMC-5-1.pml:21 (state 1)  [wantQ = 1]
spin: trail ends after 8 steps
#processes: 2
                wantP = 0
                wantQ = 1
  8:    proc  1 (Q:1) SMC-5-1.pml:22 (state 7)
  8:    proc  0 (P:1) SMC-5-1.pml:7 (state 11)
2 processes created
```

- it marks the start of the cycle with **<<<<<START OF CYCLE>>>>>**
- the last state shown is the same as the one just before that marker

"never claims" ???

- When we run the liveness analysis we see that the SPIN output is suddenly all about "claims" and "never claims".

```
Full statespace search for:
        never claim             + (:np_:)
        assertion violations    + (if within scope of claim)
        non-progress cycles     + (fairness disabled)
        invalid end states      - (disabled by never claim)
```

- What are these?

Never Claims

- Never Claims are the mechanism used by SPIN to check LTL properties
- The state models produced by SPIN are not just simple graphs
  - They are automata, known as Büchi automata
- Büchi Automata:
  - Finite state machines
  - With criteria defined for accepting both finite and infinite languages
  - This means that accepting states can be state-loops
- LTL properties can be converted to Büchi automata
  - These can be expressed in Promela
  - They run in parallel with the Promela model defined in a `.pml` file.

Finite State Machines/Automata (FSM/FSA)

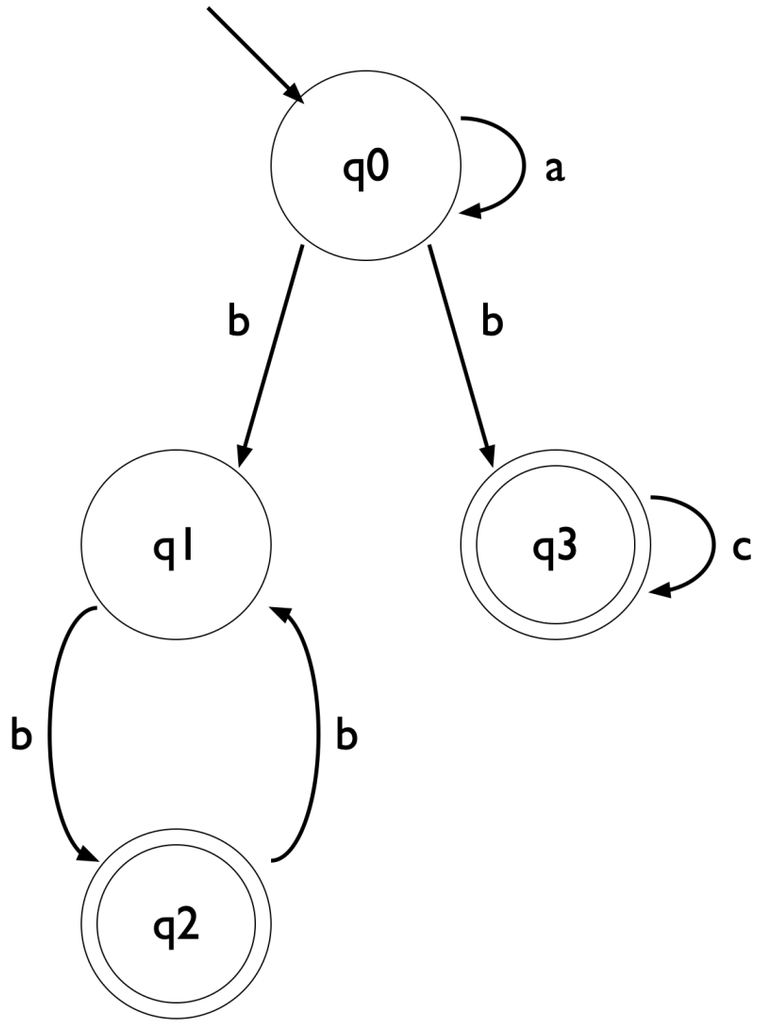
- Composed of a finite number of states
  - One is designated as the starting state
  - One, or more, are designated as accepting (final) states
- Responds to events
  - Each state responds to some of the events
  - A response "consumes" the event, and may also change the state it is in (moving to "next" state).
- Non-determinism
  - A state responding to an event may have a choice between different states to which it can change.
    - If all states have no such choice then we have a Deterministic Finite Automata (DFA)
    - If any state has such a choice, then we have a Non-deterministic Finite Automata (NDFA)

FSAs "accept" event-sequences

- Given a (N)DFA and a sequence of events, we can ask
  - Can the (N)DFA perform this sequence?
  - Does the starting event "respond" to the first event?
    - If so, does the next state "respond" to the second event?
    - and so on.
- We say that a (N)DFA "accepts" a sequence of events if there is a way to respond to all of the events such that it ends up in one of its "accepting" states.

NDFA Example

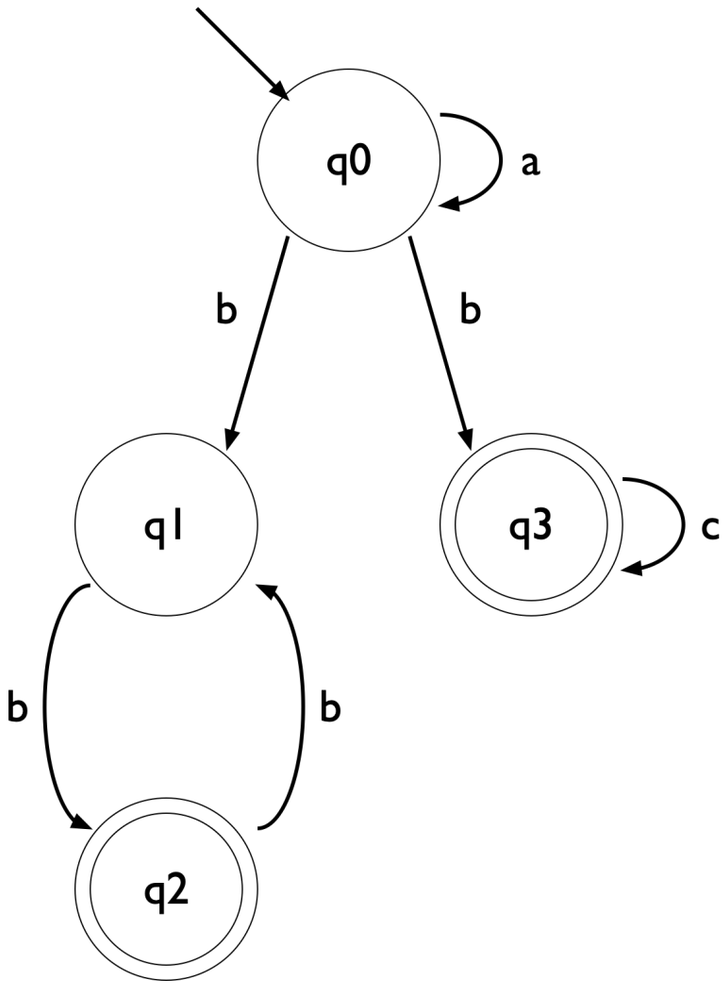
- Regular expressions are a shorthand way to describe finite sequences
  - `seq1 seq2`
    - seq1 followed by seq2
  - `seq1 + seq2`
    - either seq1 or seq2
  - `seq*`
    - zero or more repetitions of seq
  - `seq+`
    - one or more repetitions of seq
- E.g.,: `a*((bb)+ + bc*)`
  - zero or more as,
    followed by either:
    an even non-zero number of bs;
    or one b followed by zero or more cs.



Principles of the Spin Model Checker, M. Ben-Ari, Sec 8.1, pp125-7

NDFA in Promela

Any NDFA can be written in Promela




```
1  /* Copyright 2007 by Moti Ben-Ari
2     under the GNU GPL; see readme.txt */
3
4  #define LEN 5
5
6  active proctype FA() {
7    byte h;
8    byte i[LEN];
9    i[0] = 'a'; i[1] = 'a'; i[2] = 'b'; i[3] = 'b'; i[4] = '.';
10 q0: if
11     :: i[h] == 'a'  -> h++; goto q0
12     :: i[h] == 'b'  -> h++; goto q3
13     :: i[h] == 'b'  -> h++; goto q1
14     fi;
15 q1: if
16     :: i[h] == 'b'  -> h++; goto q2
17     fi;
18 q2: if
19     :: i[h] == 'b'  -> h++; goto q1
20     :: i[h] == '.'  -> goto accept
21     fi;
22 q3: if
23     :: i[h] == 'c'  -> h++; goto q3
24     :: i[h] == '.'  -> goto accept
25     fi;
26 accept:
27     printf("Accepted!\n");
28     // assert(false)
29 }
```

Applying SPIN to NDFA

- We can simulate the example,
  - **`spin SMC-8-1.pml`**

    but because the first `b` results in either `q1` or `q3` being chosen non-deterministically, we are not guaranteed to get the accepting outcome
- We can verify it
  - spin -run SMC-8-1.pml
    Here we have only 1 process, so we get a "timeout" rather than any deadlock - not helpful.
- We can "falsify" it.
  - We want to see if we can get to **`Accepted!`** message, so **`assert(false)`** just afterwards !?!?!
  - A verification run will find the only "failure" in the system (**`false`**), and generate the sequence of steps leading through label "**`accept:`**"

The “flip side” of counter-examples

- SPIN is designed to exhaustively check a model to see if any desired property is in fact false.
  - If it finds such a failure, it generates a counter-example - sequence of moves leading to failure.
    If it doesn’t, it reports that all is well.
- We can use SPIN to find solutions to problems too.
  - We design our model so that:
    - (a) there is a point in the code that corresponds to having a solution, or
      (b) we specify a property that is true when a solution is found.
  - We then:
    - (a) add in an **`assert(false)`** after the solution point in the code, or
      (b) specify the negation of the property
  - SPIN will then, eventually, find the **`assert(false)`**, or where the negated property is false, and report this as a failure with a “counter-example” that leads to that point - the solution !

How SPIN handles Linear Temporal Logic

- For every LTL predicate, there exists an NDFA that accepts any event sequence that satisfies that predicate.
- Because SPIN is looking for failures, it needs an NDFA that accepts any event sequence that ***does not*** satisfy that predicate.
- So we generate the NDFA for the ***negation*** of the predicate, which is then represented as a process that can be written in Promela.
  - This process is called a "never claim"
- This NDFA is then run in parallel with the rest of the Promela model.
  - If it ever enters an "accepting" state, then it has found a violation of the predicate
  - It also signals a failure if an **`assert()`** statement fails.

Writing LTL in Promela

- According to the books on the reading list:
  - The syntax of LTL in Promela only has variables, and the logical operators.
    - It does not have expressions (boolean or otherwise), such as x < 1 or y = z + 42.
  - Instead we need to use #define to link a variable to such expressions.
  - So `[]((x==42) -> <>(y==99))`

    has to be written as something like this:

```
#define answer (x==42)
#define balloons (y==99)
[](answer -> <>balloons)
```

- However since version 6 of Spin:
  - We can simply write `[]((x==42) -> <>(y==99))`

From LTL to never claims

- Early versions of Promela/SPIN required the modeller to work out never claims by hand
- Then a version came along that could translate an LTL predicate into a never claim.

```
Promela> spin -f 'p'
never  {    /* p */
accept_init:
T0_init:
        do
        :: atomic { ((p)) -> assert(!((p))) }
        od;
accept_all:
        skip
}
```

```
Promela> spin -f '<>p'
never  {    /* <>p */
T0_init:
        do
        :: atomic { ((p)) -> assert(!((p))) }
        :: (1) -> goto T0_init
        od;
accept_all:
        skip
}
```

The output of spin -f '….' can be piped into a file which is then included in the Promela model file.

```
Promela> spin -f '[]p'
never  {    /* []p */
accept_init:
T0_init:
        do
        :: ((p)) -> goto T0_init
        od;
}
```

- Current versions allow LTL predicates to be written directly in the Promela file

Example: a bad mutex solution

- Example of an attempt to solve mutual exclusion
  - It ensures only **`P()`** or **`Q()`** in the critical section at one time
  - Unfortunately, it can deadlock!
    - Easily demonstrated by **`spin -run`**.
  - How can we use LTL to verify the exclusion property?

Principles of the Spin Model Checker,
M. Ben-Ari, Sec 4.3, p52

```
1  /* Copyright 2007 by Moti Ben-Ari
2     under the GNU GPL; see readme.txt */
3
4  bool wantP = false, wantQ = false;
5
6  active proctype P() {
7    do ::  wantP = true;
8           !wantQ;
9           // critical section
10          wantP = false
11   od
12 }
13
14 active proctype Q() {
15   do ::  wantQ = true;
16          !wantP;
17          // critical section
18          wantQ = false
19   od
20 }
```

mutual exclusion using LTL

- We add a counter **`critical`**
- **E**ach process:
  - increments it on entry to the critical region
  - decrements it on exit from the critical region
- We define a LTL predicate **`msafe`** that asserts that **`critical`** is always less than 2 in any state.
  - If we use **`spin -a`** we can see the never claim generated, in **`_spin_nvr.tmp`**

```
1  /* Copyright 2007 by Moti Ben-Ari
2    under the GNU GPL; see readme.txt */
3  bool wantP = false, wantQ = false;
4  byte critical;
5  active proctype P() {
6    do ::  wantP = true;
7           !wantQ;
8           critical++;
9           // in critical section
10          critical--;
11          wantP = false
12   od
13 }
14 active proctype Q() {
15   do ::  wantQ = true;
16          !wantP;
17          critical++;
18          // in critical section
19          critical--;
20          wantQ = false
21   od
22 }
23 ltl msafe { [](critical <=1) }
```

Principles of the Spin Model Checker,
M. Ben-Ari, Sec 5.3.2, pp75-78

Other ways … ?

- We have seen shared-variable concurrency
  - Threads have ***non-atomic*** read-write access to shared global variables in other threads that are part of the ***same*** Process.
  - The pthread library provides a discipline for avoiding unwanted interference
    - The Promela modelling we have done has dealt with this approach
- What if we didn't share variables?
  - How can we do this?
  - Would this make all our problems go away?

Other Ways.

- The main alternative is some form of ***message passing***
- Processes/Threads communicate by sending and receiving messages
  - All variables are kept local
- Message passing is the main mechanism used by operating systems to allow communication between Processes
  - Remember, different Processes do not (usually) share any memory.
- Message Passing is the only way to go when doing any networking code
  - Different computers, far apart, cannot share main memory!

To Synchronise or not to Synchronise?

- There are two forms of message passing: synchronous, and asynchronous
- Synchronous:
  - both sender and receiver wait until the communication has completed
  - If one thread does a send/receive, but another never does a receive/send, then that thread will deadlock!
  - Also known as "Rendezvous Communication"
- Asynchronous:
  - the sender returns immediately, while the message is buffered somewhere
  - the receiver will wait for a message if necessary
  - again the only way to efficiently perform networking
- Hybrid: it is possible to mix the two

Message Passing in Promela

- Distributed (network) systems consist of nodes connected by communication channels
  - Internet: computers connected by wires, optical fibres, wireless, satellite, and networking hardware running network and communication protocols
- Promela can model this
  - Nodes/Computers are modelled by Promela Processes
  - Network Communication is modelled using Promela Channels

Channels in Promela

- Declaring a channel: **`chan`** `ch` **`=`** **`[`**`capacity`**`]`** **`of`** **`{`** `typename, …, typename` **`}`**
  - capacity is size of (hidden) buffer - can be zero, or positive
  - `typename` cannot be an array type, but can be a struct that contains an array
  - `typename` can be **`chan`** - so we can send/receive communication channels !
- Writing/Sending to a channel: `ch ! expr, …, expr`
  - the number and type of `expr` must match the types in the declaration for `ch`
- Reading/Receiving from a channel: `ch ? var, …, var`
  - the number and type of `var` must match the types in the declaration for `ch`

Simple example: Server + 2*Client

PSMC, Listing 7.1, p107

- Channel `request` has no buffer
  - Synchronous communication
  - Only time in Promela that two threads perform a step at the exact same time (one does a send-step, the other does a receive step).
- Server loops forever
  - waiting for input on channel request
- Clients make one request and stop.
  - both write to the same channel

```
1  /* Copyright 2007 by Moti Ben-Ari
2     under the GNU GPL; see readme.txt */
3
4  chan request = [0] of { byte };
5
6  active proctype Server() {
7    byte client;
8  end:
9    do
10     ::  request ? client
11           -> printf("Client %d\n", client);
12   od
13 }
14
15 active proctype Client0() {
16   request ! 0;
17 }
18
19 active proctype Client1() {
20   request ! 1;
21 }
```

Buffered Channels

- If channel capacity is non-zero, then we can have asynchronous communication
  - Send puts stuff in buffer, if not full
  - Receive takes stuff, if not empty
- There are builtin predicates to check if a channel buffer is full, empty:
  - **`full`**, **`empty`**, **`nfull`**, **`empty`**
  - e.g. **`empty`**`(mychan)` is executable/true if the buffer for mychan has no messages,

```
/* Copyright 2007 by Moti Ben-Ari under the GNU GPL; see readme.txt */

chan request = [2] of { byte, chan};
chan reply[3] = [2] of { byte };

bool waiting = false;
/* Verify []!waiting to show clients waiting */

active [2] proctype Server() {
  byte client;
  chan replyChannel;
  do
  :: empty(request) ->
       printf("No requests for server %d\n", _pid)
  :: request ? client, replyChannel ->
       printf("Client %d processed by server %d\n", client, _pid);
       replyChannel ! _pid
  od
}

active [3] proctype Client() {
  byte server;
  do
  :: full(request) ->
       waiting = true;
       printf("Client %d waiting for non-full channel\n", _pid)
  :: request ! _pid, reply[_pid-2] ->
       reply[_pid-2] ? server;
      printf("Reply received from server %d by client %d\n", server, _pid)
  od
}
```

Shared Variable vs. Message Passing

- See https://wiki.c2.com/?MessagePassingConcurrency
- See https://wiki.c2.com/?SharedStateConcurrency