Campus | B5 7mm×30行 40页

Real-time Computing and Embedded Systems

Campus®

B5 点线本 | 7mm×30行 | 40页

KOKUYO

No.

Date

Introduction to real-time systems (pdf 19)

Three programming languages : ① C/Real-Time POSIX ② Real-Time Specification for Java (RTSJ) ③ Ada

#1.1 Definition of a real-time system

Any information processing activity or system which has to respond to externally generated input stimuli within a finite and specific period. (Young, 1982)

A real-time system is a system that is required to react to a stimuli from the environment (including the passage of physical time) within time intervals dictated (affected) by the environment (Randell et al, 1995)

Embedded computer systems: In hard or soft real-time system, the computer is usually interfaced directly to some physical equipment and is dedicated to monitoring or controlling the operation of that equipment

Characteristics of real-time systems

[Real-time control facilities]

① Specify times at which actions are to be performed
② Specify times by which actions are to be completed
③ Support repeating (period or a period) work
④ Control the jitter on output and input operations
⑤ Respond to situations where not all of the timing requirements can be met
⑥ Respond to situations where the timing requirements are changed dynamically

[Concurrent control of separate system components] #重

Distributed and multiprocessor embedded systems

input data
required computation with no external interactions
output data

} Sufficient form for a reactive control systems with hard timing constraints, but relatively straightforward behaviour.

[Low-level programming]

Monitor sensors and control actuators for a wide variety of real-world devices.

Since real-time systems are time-critial, efficiency of implementation will be more important ~~that~~ than in other systems.

The benfits of using a high-level language is that it enables the programmer to abstract away from implementation details while for embedded computer systems programmers, he or she must be constantly concerned with the cost of using particular language features.

[Low-level is faster than the high-level]

[Support for numerical computation]

Impliementation of control systems.

Summary

Real-time system : any information processing activity or system which has to respond to externally generated input stimuli ~~which~~ within a finite and specified delay

Aspects of real-time systems

- Real-time
  - Classification
    - Critically of deadlines
      - Hard
      - Soft
      - Firm 固件
    - Role of time
      - Time-aware
      - Reactive
  - Temporal requirements
    - Deadline / latency
    - Input / output jitter
    - Periodic / Sporadic / Aperiodic 散的
  - Structure
    - Time-triggered
    - Event-triggered
  - Characteristics
    - Real-time facilities
    - Concurrent
    - Numerical computations
    - Interaction with hardware devices
    - Efficient and predictable
    - ~~Reali~~ Reliable and safe
    - Large and complex

~~Hard~~ real-time systems
→ It is absolutely imperative (不可缺少) that responses occur within the specified deadline.

Soft real-time systems
→ Response times are important, but the system will still function correctly if deadlines are occasionally missed.

General criteria for a real-time language design :

Security　Readability　Flexibility　Simplicity

Portability　~~Efficient~~　Efficiency

No.

Date

Free

Simple User Interface

▶ app > res > layout > activity_main.xml

Subclass of ViewGroup

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"      } Specify the size
    android:layout_height="match_parent"     } ← Match the width or height of the parent view
    android:orientation="horizontal">        ← "horizontal" or "vertical"
```

[Add a Text Field]

```
</LinearLayout>
```

"+" is required when you're defining a resource ID for the first time
unique identifier for the view
is required when you're referring to any resource object from XML { id / resource type / resource name } edit_message

```
<EditText android:id="@+id/edit_message"
    android:layout_width="wrap_content"   ← Specify the view should be only as big as needed to fit the contents of view
    android:layout_height="wrap_content"
    android:hint="@string/edit_message" />   ← Default String to display when the text field is empty
```

[Add String Resources] ▶ res > values > strings.xml

```
<resources>
    <string name="app_name">My First App</string>
    <string name="edit_message">Enter a message</string>
    <string name="button_send">Send</string>
</resources>
```

[Add a Button]

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button_send" />
    android:onClick="sendMessage" />
```

[Respond to the Send Button] res > layout > activity_main.xml

java > com.example.myfirstapp > MainActivity.java

☆ "Alt + Enter" to import class

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    public void sendMessage(View view) {
        // Do something in response to button
    }
}
```

Main Function

變variable

```
public final static String EXTRA_MESSAGE = "xxxx";
```

Intent is an object that provides runtime binding between separate components (e.g. two activities)
This intent starts another activity

[Build an Intent]

```
Intent intent = new Intent(this, DisplayMessageActivity.class);
EditText editText = (EditText) findViewById(R.id.edit_message);
String message = editText.getText().toString();
intent.putExtra(EXTRA_MESSAGE, message);   ← The putExtra() method adds the EditText's value to the intent
startActivity(intent);   ← Starts an instance
```

An intent can carry data types as key-value pairs (鍵值對)
Your key is a public constant EXTRA_~~EDITT~~MESSAGE
Next activity uses the key to retrieve the next value

No.

Date

① Create the class DisplayMessageActivity.java with an implementation of the required onCreate() method
② Create the corresponding layout file activity_display_message.xml
③ Adds the required <activity> element in AndroidManifest.xml.
Activity Name : DisplayMessageActivity

[Creat the Second Activity] right-click the "app", New → Activity → Empty Activity

DisplayMessageActivity.java

```
public class DisplayMessageActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_display_message);

        Intent intent = getIntent(); ← grabs the intent that started the activity
        String message = intent.getStringExtra(MainActivity.EXTRA_MESSAGE); ← retrieves the data from the fist activity
        TextView textView = new TextView(this);
        textView.setTextSize(40);
        textView.setText(message);

        ViewGroup layout = (ViewGroup) findViewById(R.id.activity_display_message);
        layout.addView(textView);
    }
}
```

Campus

No.

Date

Introduction

Syllabus

Issues and concepts
- Definition of real-time
- Temporal and event determinism
- Architecture review and interfacing
- Interrupts, traps and events
- Response times and latency
- Real-time clocks

Application domains
- DSP
- Safety critial
- Small embedded
- Large-scale distributed

Low-level programming for real-time
- I/O
- Concurrency: memory models and synchronisation primitives (同步原语)
- Monitors / condition variables
- Semaphores (信号)
- Optimistic scheduling
- ARM and Intel assembly language, integration with C
- Architectures issues, memory models.

Scheduling
- RMS
- EDF
- priority inversion
- Time triggered

Operating systems
- Protected modes, virtual memory
- Device drivers
- Internet of things: TinyOS & Contiki
- FreeRTOS
- Real-time Linx

Language in real-time
- C and C++ standards: MISRA

Correctness
- concurrency issues
- process algebras
- model checkers, temporal logic

No.

Date

What is a real-time system?

A real-time system is any information processing system which has to respond to externally generated input stimuli within a finite and specified period.

- the correctness depends not only on the logic results but also the time it was delivered.
- failure to respond is as bad as the wrong response

Compare with conventional systems

Partial correctness: If a result is returned, it is correct. Established using invariants (不变式)

Total correctness: Will terminate and returns a correct result. Troubled by the halting problem.

[halting problem (停机问题)] is the problem of determining, from a description of an arbitrary program and input, whether the program will finish running or continue to run forever.

Real-time correctness: will return a correct result by the deadline

Example of partial correctness

```
Integer sumsq (const Integer n){
    assert (n>=0);
    return (n==0)?0: sumsq(n-1) + n*n;}
```

```
Integer sumsq (const Integer n){
    assert (n>=0);
    return (n==0)?0: sumsq(n+1) - (n+1)*(n+1);}
```

Obviously returns the correct answer, if it returns at all.
No variables are changed, so all program values behave like algebraic unknowns
Integer which does not overflow (fake type)

Steps to (non-Real-Time) correctness

Use ~~idem~~ idealised mathematical types to simplify the reasoning.

Establish partial correctness

Establish that the function will terminate. The second example does not, if $n > 0$.

Arrange to cope with any problems from C's restricted int implementation.

No.

Date

Free RTOS

Structure of an embedded application

Standard bootstrap (引导程序) code : possibly tuned for application

Compiler libraries (e.g. for floating point)

Standard C libraries : probably modified for platform

Standard protocol libraries (e.g. TCP/IP, USB)

Processor libraries : to drive on-chip devices

Board libraries : to drive on-board devices

System management

How do you the protocols and devices share the processor?
Can run "bare metal", e.g. using a timing loop.
Can instead use a Real Time Operating System.

So also need:
- Standard RTOS
- Processor support for RTOS
- ~~Applifi~~ Application configuration for RTOS
- Adaptor layer RTOS/Libraries (e.g. Freescale Kinetis)

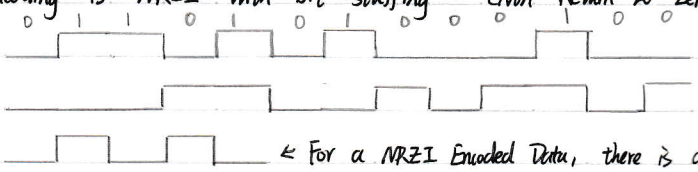
No.

Date

USB

Bit Layer

Data encoding is NRZI with bit-stuffing (Non Return to Zero Inverted)



← For a NRZI Encoded Data, there is always a Sync Pattern

NRZI 无自同步特性，在发送Data前，会发送加上同步头，内容为0101的方波，从而使接受端获取发送端的频率

Protocol layer

All data is assembled into packets:

Sync: synchronization bits: NRZI-encoded 0s

Packet ID: packet type

Address: 0-127 USB addresses (sender or receiver)

Endpoint: 0-15 Endpoints within device

Frame number: You wouldn't want to use the same data twice

Data: at most 8 bytes for low-speed USB

CRC: error detecting code

End of packet

Four classes of packets

PID types	PID Name	PID <3:0>
Token	OUT	0001
	IN	1001
	SOF	0101
	SETUP	1101
Data	DATA0	0011
	DATA1	1011
	DATA2	0111
	MDATA	1111
Handshake	ACK	0010
	NAK	1010
	STALL	1110
	NYET	0110
Special	PRE	1100
	ERR	1100
	SPLIT	1000
	PING	0100
	Reserved	0000

No.

Date

Transactions

The universal serial bus specification defines four transfer / endpoint types.

① Control transfer
② Interrupt transfer
③ Isochronous transfer ← 同步/实时 传输
④ Bulk transfer

Not for low-speed USB

[Interrupt transfer]

Under USB, if a device requires the attention of the host, it must wait until the host polls (调查) it before it can report that it needs urgent attention

Interrupt transfer are typically non-periodic, small device "initiated" communication requiring bounded latency. An interrupt request is queued by the device until the host polls the USB device asking for the data.

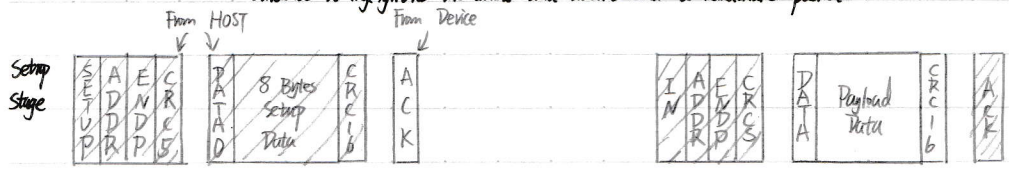
A single IN or OUT transaction makes up an interrupt transfer.

[Control transfer]

Control transfer are typically used for command and status operations. They are essential to set up a USB device with all enumeration function (枚举函数) being performed using control transfers. They are typically bursty, (猝发) random packets which are initiated by the host and use best effort delivery.

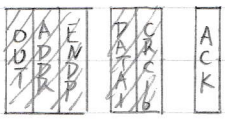
A control transfer can have up to three stages.

① Setup Stage: is where the request is sent. This consists of three packets. The setup token is sent first which contains the address and endpoint number. The data packet is sent next and always has a PID type of data0 and includes a setup packet which details the type of request. The last packet is a handshake used for acknowledging successful receipt or to indicate an error. If the function successfully receives the setup data (CRC and PID etc Ok) it responds with ACK, otherwise it ignores the data and doesn't send a handshake packet.



② Data Stage: consists of one or multiple IN or OUT transfers. The setup request indicates the amount of data to be transmitted in the stage. If it exceeds the maximum packet size, data will be sent in multiple transfers each being the maximum packet length except for the last packet.

③ Status Stage: reports the status of the overall request and this once again varies due to direction of transfer.



Endpoints

Every device is required to provide endpoint 0 for control and enumeration: the control transfers are bidirectional.

Other endpoints can transfer data using interrupt transfers. These endpoints are unidirectional. A typical keyboard would have one device → host endpoint for keyboard data and one host → device endpoint for keyboard lights. Low-speed USB cannot do isochronous or bulk transfers. So no COM: ports or memory sticks.

Device startup: The host uses control transfers to endpoint 0 to learn the capabilities of the device. The first implementation information is the maximum packet size.

After that, there are a range of descriptions. They lead to the setting up of logical devices and additional endpoints.

No.

Date

DSP

Review : Processor Classes

General Purpose - high performance
- X64, IBM Power ; ARM Cortex A ; MIPS
- Used for general purpose software
- Heavy weight OS - UNIX, NT
- Workstations, PC's

Embedded processors and processor cores
- ARM Cortex M4, M7, R ; TMS320
- Single program
- Lightweight, often realtime OS
- DSP support
- Cellular phones, consumer electronics (e.g. CD players)

Microcontrollers
- ARM Cortex M0, M0+ ; MSP430 ; AVR ; PIC
- Extremely cost sensitive
- Small word size - 8 bit common.
- Highest volume processors by far
- Automobiles, toasters, thermostats, ...

DSP introduction

Digital Signal Processing : application of mathematical operations to digitally represented signals.

Signals represented digitally as sequences of samples.

Digital signals obtained from physical signals via transducers and analogue-to-digital convertors (ADC)

Digital signals converted back to physical signal via digital-to-analogue convertors (DAC)

Digital Signal Processing (DSP) : Electronic system that processes digital signals

Common DSP algorithms and applications

Applications - Instrumentation and measurement
- Communications
- Audio and Video processing
- Graphics, image enhancement, 3-D rendering
- Navig Navigation, radar, GPS
- Control - robotics, machine vision, guidance

Algorithms
- Frequency filtering in time domain : FIR and IIR
- Spectral method : FFT
- Correlation
- Adaptive filtering : Kalman
- Machine learning

What do DSPs need to do well?

Most DSP tasks require :
- Repeatitive numeric computations
- Attention to numeric fidelity (保真度)
- High memory bandwidth, mostly via array accesses
- Real-time processing

DSPs must perform these tasks efficiently while minizing :
- Cost
- Power
- Memory use
- Development time

No.

Date

Scheduling

Scheduling

Topics
- Simple process model
- The cyclic executive approach
- Process-based scheduling
- Utilization-based scheduling schedulability tests
- Response time analysis for FPS and EDF
- Worst-case execution time
- Sporadic and aperiodic processes
  不定时发生的 非周期性的

Goal
- To understand the role that scheduling and schedulability analysis plays in predicting that real-time applications meet their deadlines.
- Process systems with D<T
- Process interactions, blocking and priority ceiling protocols.
- An extendible process model
- Dynamic systems and on-line analysis
- Programming priority-based systems

In general, a scheduling scheme provides two features:
- An algorithm for ordering the use of system resources (in particular the CPUs)
- A means of predicting the worst-case behaviour of the system when the scheduling algorithm is applied.

The prediction can then be used to confirm the temporal requirements of the application

Simple Process Model

The application is assumed to consist of a fixed set of processes
All processes are periodic, with known periods
The processes are completely independent of each other
All system's overheads, context-switching times and so on are ignored (i.e, assumed to have zero cost)
All processes have a deadline equal to their period (that is, each process must complete before it is next released)
All processes have a fixed worst-case execution time.

Standard Notation

B	Worst-case blocking time for the process (if applicable)
C	Worst computation time (WCET) of the process
D	Deadline of the process
I	The interference time of the process
J	Release jitter of the process
N	Number of processes in the system
P	Priority assigned to the process (if applicable)
R	Worst-case response time of the process
T	Minimum time between process releases (process period)
U	The utilization of each process (equal to C/T)
a-z	The name of process.

$\frac{dx^2}{dx} = 2x$

Two process $R_2 = C_2 + \lceil \frac{R_2}{T_1} \rceil \cdot C_1$ $\quad w_n = C_2 + \lceil \frac{w_{n-1}}{T_1} \rceil \cdot C_1$ $\quad$ In the most case $T_1 = C_2 + C_1$

$w^0 = C_2$ $\quad \Rightarrow T_2 = C_2 + 2C_1$

$w_n = C_2 + \lceil \frac{C_2}{T_1} \rceil C_1$

For two process $\begin{cases} T_1 = C_1 + C_2 \\ T_2 = 2C_1 + C_2 \end{cases}$

$\Rightarrow \lambda = \frac{T_2}{T_1}$

$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} = \frac{T_2 - T_1}{T_1} + \frac{2T_1 - T_2}{T_2} = \frac{T_2}{T_1} + 1 + \frac{2T_1}{T_2} - 1 = \lambda + \frac{2}{\lambda} - 2$

$\frac{dy}{dt} = -\frac{2}{\lambda^2} + 1 = 0 \Rightarrow \lambda^2 = 1$

$\lambda = 2^{\frac{1}{2}}$ $\quad U = 2^{\frac{1}{2}} + 2^{\frac{1}{2}} - 2 = 2(2^{\frac{1}{2}} - 1)$

No.

Date

Cyclic Executives

One common way of implementing hard real-time systems is to use a cyclic executive
Here the design is concurrent but the code is produced as a collection of procedures.
Procedures are mapped onto a set of minor cycles that constitute the complete schedule (or major cycle)
{ minor cycle dictates the minimum cycle time
{ Major cycle dictates the maximum cycle time

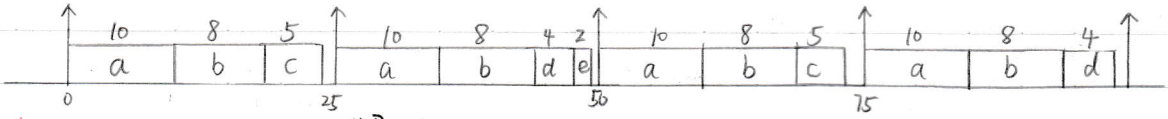
Has the advantage of being fully deterministic

Consider a Process set

Process	Period, T	Computation Time, C
a	25	10
b	25	8
c	50	5
d	50	4
e	100	2

[OS]
The cyclic executive is, essentially, a table of procedure calls, where each procedure represents part of the code for a 'task'
← The complete table is known as the major cycle, it typically consists of a number of minor cycles each of fixed duration.
E.g. four minor cycles of 25ms duration would make up a 100ms major cycle.
During execution, a clock interrupt every 25ms will enable the scheduler to loop through the four minor cycles.

↳ Possible mapping onto the cyclic executive which illustrates the job that the processor is executing at any particular time:




```
loop
  wait for interrupt;
  procedure_for_a; p_b; p_c;
  wait for interrupt;
  p_a; p_b; p_d; p_e;
  wait for interrupt;
  p_a; p_b; p_c;
  wait for interrupt;
  p_a; p_b; p_d;
end loop;
```

Property

No actual processes exist at run-time; each minor cycle is just a sequence of procedure calls
The procedures share a common address space and can thus pass data between themselves. This data does not need to be protected (via a semaphore, for example) because concurrent access is not possible.
All 'task' or 'process' periods must be a multiple of the minor cycle time.

The final property represents one of the major drawbacks of the cyclic executive approach

Issues with cyclic executives

the difficulty of incorporating sporadic tasks (impossible);
the difficulty of incorporating tasks with long periods. the major cycle time is the maximum period that can be accommodated without secondary schedules (that is, a procedure in a major cycle that will call a secondary procedure every N major cycles)
the difficulty of actually constructing the cyclic executive — NP-hard problem
any 'task' or 'procedure' with a sizeable computation time will need to be split into a fixed number of fixed sized procedures (this may cut across the structure of the code from a software engineering perspective, and hence maybe error-prone) 易出错
More & flexible scheduling methods are difficult to support
Determinism is not required, but predictability is.

No.

Date

Task-based scheduling

Fixed-Priority Scheduling (FPS)

- This is the most widely used approach and is the main focus of this course.
  Each task has a fixed, static, priority which is computed pre-run-time
- The runnable tasks are executed in the order determined by their priority.
- In realtime systems, the 'priority' of a task is derived from its temporal requirements, not its importance to the correct functioning of the system or its integrity.

Earliest Deadline First (EDF) Scheduling

- Here the runnable tasks are executed in the order determined by the absolute deadlines of the tasks;
- The next task to run being the one with the shortest (nearest) deadline.
- Although it is usual to know the relative deadlines of each task (e.g. 25ms after release), the absolute deadlines are computed at run-time, and hence the scheme is described as dynamic.

Value-Based Scheduling (VBS)

- If a system can become overloaded (current utilization greater than 100%) then the use of simple static priority or deadlines is not sufficient
- A more adaptive scheme is needed. This often takes the form of assigning a value to each task and employing an online value-based scheduling algorithm to decide which task to run next.

Scheduling characteristics (ascribe scheduling test)

A schedulability test is defined to be sufficient if positive outcome guarantees that all deadlines are always met.

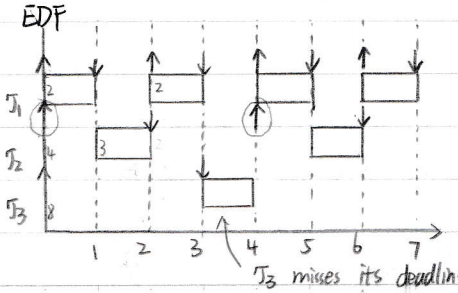
A test can also be labelled as necessary if failure of the test will indeed lead to a deadline miss at some point during the execution of the system

Preemption and non-preemption

With priority-based scheduling, a high-priority task may be released during the execution of a lower-priority one.
In a preemptive scheme, there will be an immediate switch to the higher-priority access
With non-preemption, the lower-priority process will be allowed to complete before the other executes.
Preemptive schemes enable higher-priority processes to be more reactive, and hence they are preferred
Alternative strategies allow a lower priority process to continue to execute for a bounded time
These schemes are known as deferred preemption or cooperative dispatching

Example: scheduling using EDF

Task	C_i	D_i	T_i	U
T_1	1	1	2	0.5
T_2	1	2	4	0.25
T_3	1	3	8	0.125



KOKUYO

No.

Date

Fixed-priority scheduling (FPS)

With the straightforward model outlined above, there exists a simple optimal priority assignment scheme for FPS known as rate monotonic priority assignment

Each task is assigned a (unique) priority based on its period. The shorter the period, the higher the priority $\begin{cases} T_i < T_j \\ P_i > P_j \end{cases}$

This assignment is optimal in the sense that if any process set can be scheduled (using pre-emptive priority-based scheduling) with a fixed-priority assignment scheme, then the given process set can also be scheduled with a rate monotonic assignment scheme

Task	Period, T	Priority, P
a	25	5
b	60	3
c	42	4
d	105	1
e	75	2

(The higher the integer, the greater the priority)

N	Utilization bound
1	100%
2	82.8%
3	78.0%
4	75.7%
5	74.3%
10	71.8%

← $N \cdot (2^{\frac{1}{N}} - 1)$

$N \to \infty$, $U \leq 0.693$

Utilization-based schedulability tests for FPS

Very simple schedulability test for FPS which although not exact.

By considering only the utilization of the task set, a test for schedulability can be obtained (when the rate monotonic priority ordering is used). If the following condition is true then all N tasks will meet their deadlines (not that the summation calculates the total utilization of the ta

$$\sum_{i=1}^{N} \left(\frac{C_i}{T_i}\right) \leq N \cdot (2^{\frac{1}{N}} - 1)$$

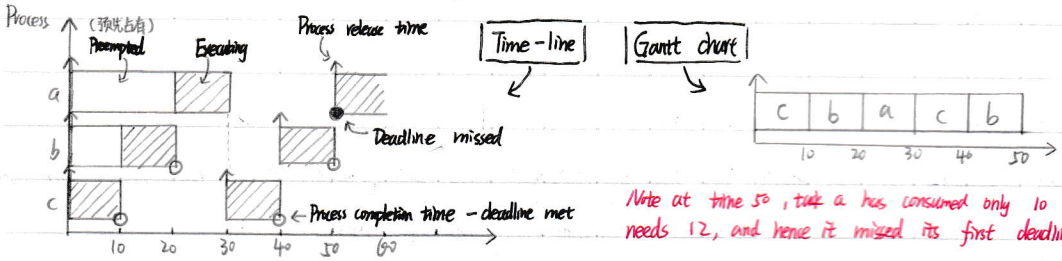
For large N, the bound asymptotically (渐近) approaches 69.3%. Hence any task set with a combined utilization of less than 69.3% will always be schedulable by a preemptive priority-based scheduling scheme, with priority assigned by the rate monotonic algorithm.

Task	Period, T	Computation time, C	Priority, P	Utilization, U
a	50	12	1	0.24
b	40	10	2	0.25
c	30	10	3	0.33

$\left(U = \frac{C_i}{T_i}\right)$

⇐ Combined utilization is 0.82 (82%) > 78%

↳ This test set fails the utilization test



Note at time 50, task a has consumed only 10 ticks of execution, whereas it needs 12, and hence it missed its first deadline.

Task	Period, T	Computation time, C	Priority, P	Utilization, U
a	80	32	1	0.400
b	40	5	2	0.125
c	16	4	3	0.250

⇐ The combined utilization is 0.775 which is below the bound hence this task set is guaranteed to meet all its deadlines.

Task families

There is a general rule there. We can improve the sensitivity of the Liu-Layland test by counting task families, not individual ta

A task family is a set of tasks whose periods are all integer multiples of the shortest period in the family

In process C, there is only one task family, so the utilisation bound is 1

周期都是周期最小任务的整数倍 (20×1, 20×2, 20×4) (c, b, a)

Task	T	C	P	U
a	80	40	1	0.50
b	40	10	2	0.25
c	20	5	3	0.25

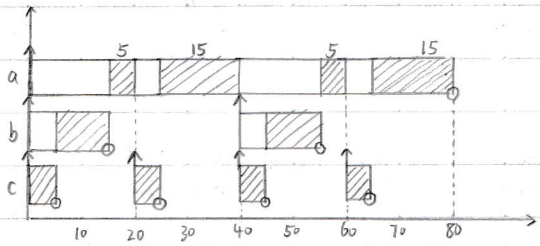
This is again a three-task system, but the combined utility is now 100%, so it clearly fails the test. At run-time, however, the behaviour seems correct, all deadlines are meet up to time 80

Hence the task set fails the test, but at run-time does not miss a deadline.

Therefore, the test is sufficient but not necessary

↳ failure of the test will indeed lead to a deadline miss

↳ positive outcome guarantees that all deadlines are always met



If a task set passes the test, it will meet all deadlines
if it fails the test, it may or not fail at run-time

This utilization-based test is that it only supplies a simple yes/no answer.
It does not give any indication of the actual response times of the tasks

Improved utilization-based test

$$\prod_{i=1}^{N} \left(\frac{C_i}{T_i} + 1\right) \leq 2$$

[Bini et al., 2003]

↑ Sometimes called Hyperbolic Bound

Task	T	C	P	U
a	76	32	1	0.421
b	40	5	2	0.125
c	16	4	3	0.250

The combined utilization is 0.796 > 0.78

But $1.421 * 1.125 * 1.25 = 1.998 < 2$

Task set is schedulable by the Bini et al. test

Criticism Criticism of Utilization-based Tests

Time complexity (时间复杂度)

Not exact, Not General, But it is O(N) The test is sufficient but not necessary

Utilization-based Test for EDF

$$\sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1$$

Superior to FPS as it can support high utilizations
FPS is easier to implement as priorities are static
EDF is dynamic and requires a more complex run-time system which will have higher overhead
It is easier to incoporate processes without deadlines into FPS: giving a process an arbitrary deadline is more artifical.
It is easier to incoporate other factors into the notion of property than it is into the notion of deadline.

During overload

FPS is more predictable: Low priority process miss their deadline first.

EDF is unpredictable: a domino effect can occur in which a large number of processes miss deadlines.

No.

Date

response time：等级低的任务在Deadline之前所需要完成任务的最短时间 $R_i < D_i$ (if schedulable)

Response time analysis (RTA) for FPS (Necessary and sufficient)

The utilization-based tests for FPS have two significant drawbacks: they are not exact, and they are not really applicable to a more general task model

This section provides a different form of test. The test is in two stages

① An analytical approach is used to predict the worst-case response time (R) of each task

These values are then compared, trivially (现在), with the task deadlines. This required each task to be analysed individually.

For the highest-priority task, its worst-case response time will equal its own computation time (that is, $R=C$). Other tasks will suffer interfere from high-priority tasks; this is the time spent executing high-priority tasks when a low-priority task is runnable.

So for a general task i: $R_i = C_i + I_i$

where I_i is the maximum interference that task i can experience in any time interval $(t, t+R_i)$ 时间

The maximum interference ~~that task i can experience in any time~~ obviously occurs when all higher-priority tasks are released at the same time as task i (that is, at a critical instant). Without loss of generality, it can be assumed that all tasks are released at time 0. Consider one task j of higher priority than i. Within the interval $[0, R_i)$, it will be released a number of times (at least one). A simple expression for this number of releases is obtained using a ceiling function: (上限)

Number of releases $= \lceil \frac{R_i}{T_j} \rceil$ $\lceil \frac{R_i}{T_j} \rceil$

The ceiling function ($\lceil\ \rceil$) gives the smallest integer greater than the fractional number on which it acts. So the ceiling of $\frac{1}{3}$ is 1, of 6.5 is 2, and of 6/3 is 2. The definitions of the ceilings of negative values need not be considered.

{ ceiling function $\lceil\ \rceil$ greater than the fractional number $1.5 \to 2$
{ floor function $\lfloor\ \rfloor$ smaller than the fractional number $1.5 \to 1$

So, if R_i is 15 and T_j is 6 then there are three releases of task j (at times 0, 6 and 12 除不尽). Each release of task j will impose an interference of C_j. Hence:

Maximum Interference $= \lceil \frac{R_i}{T_j} \rceil \cdot C_j$

If $C_j = 2$ then in the interval $[0, 15)$ there are 6 units of interference. Each of task of higher priority is interfering with task i, and hence:

$I_i = \sum_{j \in hp(i)} \lceil \frac{R_i}{T_j} \rceil \cdot C_j$ where $hp(i)$ is the set of higher-priority tasks (than i)

As $R_i = C_i + I_i$, we have $R_i = C_i + \sum_{j \in hp(i)} \lceil \frac{R_i}{T_j} \rceil \cdot C_j$

Although the formulation of the interference equation is exact, the actual amount of interference is unknown as R_i is unknown (it is the value being calculated)

As the equation above has R_i on both sides, but is difficult to solve due to the ceiling function. In general, there will be many values of R_i that from solutions to Equation. The smallest such value of R_i represents the worst-case response time for the task.

The simplest way to solve the Equation is to form a recurrence relationship (递推关系)

$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \lceil \frac{w_i^n}{T_j} \rceil \cdot C_j$

The set of values $\{w_i^0, w_i^1, w_i^2, \cdots, w_i^n, \cdots\}$ is, clearly, monotonically non-decreasing. When $w_i^n = w_i^{n+1}$, the solution to the equation has been found. If $w_i^0 < R_i$ then w_i^n is the smallest solution and hence is the value required. If the equation does not have a solution then the w values will continue to raise (this will occur for a low-priority task if the full set has a utilization greater than 100%). Once they get bigger than the task's period, T_i, it can be assumed that the task will not meet its deadline. The starting value for the process w_i^0, must not be greater than the final (unknown) R_i. As $R_i \geq C_i$ a safe starting point is C_i

The above analysis gives rise to the following algorithm for calculation response times:

```
for i in 1..N loop -- for each task in turn
   n := 0
   w_i^n := C_i
   loop
      calculate new w_i^n+1 from equation  w_i^n+1 = C_i + Σ_{j∈hp(i)} ⌈w_i^n / T_j⌉ · C_j
      if w_i^n+1 = w_i^n then
         R_i := w_i^n
         exit value found
      end if

      if w_i^n+1 > T_i then
         exit value not found
      end if

      n := n+1
   end loop
end loop
```

$R_i < D_i$

By implication, if a response time is found it will be less than T_i, and Hence less than D_i, its deadline (with simple task model $D_i = T_i$).

Task	Period, T	Computation, C	Priority, P
a	7	3	3
b	12	3	2
c	20	5	1

The highest-priority task, a, will have a response time equal to its computation time, $R_a = C_a = 3$

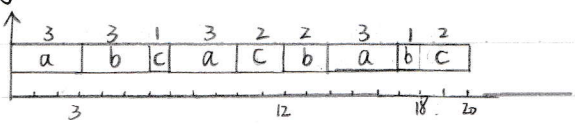
The next task will need to have its response time calculated

Let w_b^0 equal to the computation time of task b, which is $C_b = 3$ (safe starting point)

$w_b^1 = 3 + \lceil \frac{3}{7} \rceil \cdot 3 = 6$

$w_b^2 = 3 + \lceil \frac{6}{7} \rceil \cdot 3 = 6 = w_b^1 \Rightarrow$ The response time of task b has been found ($R_b = 6$)

The final task will give rise to the following calculation:



$w_c^0 = 5$

$w_c^1 = 5 + \lceil \frac{5}{7} \rceil \cdot 3 + \lceil \frac{5}{12} \rceil \cdot 3 = 11$

$w_c^2 = 5 + \lceil \frac{11}{7} \rceil \cdot 3 + \lceil \frac{11}{12} \rceil \cdot 3 = 14$

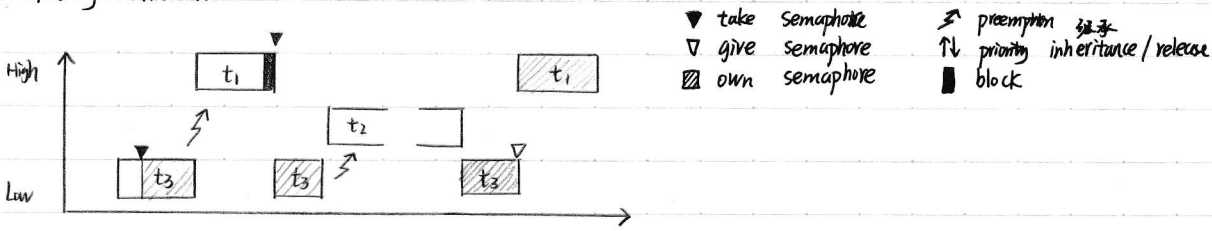
$w_c^3 = 5 + \lceil \frac{14}{7} \rceil \cdot 3 + \lceil \frac{14}{12} \rceil \cdot 3 = 17$

$w_c^4 = 5 + \lceil \frac{17}{7} \rceil \cdot 3 + \lceil \frac{17}{12} \rceil \cdot 3 = 20$

$w_c^5 = 5 + \lceil \frac{20}{7} \rceil \cdot 3 + \lceil \frac{20}{12} \rceil \cdot 3 = 20 = w_c^4 \Rightarrow R_c = 20$ ← which means it will just meet its deadline.

The response time calculations have the advantage that they are sufficient and necessary – if the task set proses the test they will meet their deadlines; if they fail the test, then, at run-time, a task will miss its deadline (unless the computation time estimation, C, themselves turn out to be pessimistic).

Priority Inversion



Priority inversion arises when a higher-priority task is forced to wait an indefinite period of time for a lower-priority task to complete. t_1, t_2, t_3 are tasks of high, medium, and low priority, respectively.

t_3 has acquired some resource by taking its associated binary guard semaphore (=进入临界保护信号)

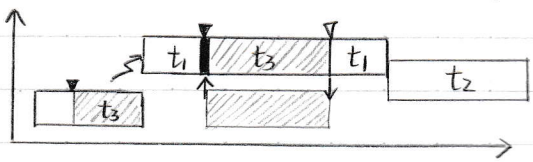
When t_1 preempts (取得优先权(=一)) t_3 and contends (竞争) for the resource by taking the same semaphore, it becomes blocked.

If we could be assured that t_1 would be blocked no longer than the time it normally takes t_3 to finish with the resource, there would be no problem because the resource cannot be preempted.

However, the low-priority task is vulnerable to preemption by medium-priority tasks (like t_2), which could inhibit (抑制) t_3 from relinquishing (放弃) the resource. This condition could persist (持续), blocking t_1 for an indefinite period of time.

Priority Inheritance

We can reduce priority inversion with a priority-inheritance algorithm. The priority-inheritance protocol assures that a task that owns a resource executes at the priority of the highest-priority task blocked on that resource.



The priority inheritance solves the problem of priority inversion by elevating the priority of t_3 to the priority of t_1 during the time t_1 is blocked on the semaphore. This protects t_3, and indirectly t_1, from preemption by t_2.

RMS . RMA

From High-Performance Embedded Computing Architectures Applications and Methodologies (Pdf 230w)

In their classic paper, Liu and Layland analyzed examples of both static and dynamic priority scheduling algorithms. Their static priority algorithm was called rate-monotonic scheduling (RMS) or rate-monotonic analysis (RMA). Their dynamic priority algorithm was known as earliest-deadline first (EDF). Their analysis made some common assumptions:

① There are no data dependencies between processes.

② Process periods may have arbitrary relationships.

③ Context-switching overhead is negligible.

④ The release time of each process is at the start of its period.

⑤ Process execution time (C_i) is fixed.

Liu and Layland defined process utilization as the sum of the utilizations of the component processes.

$$U = \sum_{1 \le i \le m} \frac{C_i}{T_i}$$

They showed that the least upper bound on utilization for a set of m tasks scheduled by RMS is

$$U = m \cdot (2^{\frac{1}{m}} - 1)$$

Liu and Layland also studied Earliest-deadline First scheduling (EDF), which they called deadline driven scheduling. Priories are assigned to processes based on the time remaining until the process's deadline. - the highest-priority process is the one that is closest to reaching its deadline. Those priorities are updated at each potential context switch.

Liu and Layland gave a feasibility condition for the schedulability of a system of processes using EDF. Given a set of n processes, let D_i be the relative deadline of process i. Then the process set must satisfy:

$$\sum_{1 \le i \le n} \frac{T_i}{\min(D_i, C_i)} \le 1 \quad \rightarrow \quad \sum_{i=1}^{n} \cdot \frac{T_i}{C_i} \le 1$$

Worst case

$$\begin{cases} C_i = T_{i+1} - T_i \\ C_n = 2T_1 - T_n \end{cases} \Rightarrow U_i = \frac{C_i}{T_i} = \frac{T_{i+1}}{T_i} - 1$$

$$U_{i+1} = \frac{T_{i+1}}{T_i}$$

$$U_{n+1} = \frac{C_n}{T_n} = \frac{2T_1}{T_n}$$

$$\Rightarrow \prod_{i=1}^{n} (U_{i+1}) = \frac{T_2}{T_1} \cdot \frac{T_3}{T_2} \cdots \frac{2T_1}{T_n} = 2$$

Thread 线程

Deadlock

In concurrent computing, a deadlock is a state in which each member of a group of actions, is waiting for some other member to release a lock.

Livelock

A livelock is similar to a deadlock, except that the state of the process involved in the livelock constantly change with regard to one another.

Synchronization

- Busy Waiting
- Hardware techniques, eg test-and-set
- Semaphores
- Dining philosophers, readers/writers

Deadlock and its avoidance.

The major difficulties associated with concurrent programming arise from task interactions. The correct behaviour of a concurrent program is critically dependent on synchronization and communication between tasks. In the widest sense, synchronization is the satisfaction of constraints on the interleaving of actions of different tasks. Communication is the passing of information from on task to another.

Inter-task communication is usually based upon either shared variables or message passing.

Shared-variables are objects to which one more than one task has access; communication can therefore proceed by each task referencing these variables when appropriate.

Message passing involves the explicit exchange of data between two tasks by means of a message that passes from one task to another via some agency (任何)

互斥

Mutual exclusion and condition synchronization

Although shared variables appear to be a straightforward way of passing information between tasks, their unrestricted use is unreliable and unsafe due to multiple update problems.

Consider two tasks updating a shared variable, X, the assignment: X:=X+1;

Most hardware this will nit be executed as an indivisible (atomic) operation (cannot be executed concurrently with any other operation involving a data item), but will be implemented in three distinct instructions:

(1) Load the value of X into some register (or to the top of the stack)
(2) Increment the value in the register by 1
(3) Store the be value in the register back to X

As the three operations are not indivisible, two tasks simultaneously updating the variable could follow an interleaving that would produce an incorrect result. Each task could load x into their registers and store the value at the same time.

A sequence of statements that must appear to be executed indivisibly is called a critical section. The synchronization required to protect a critical section is known as mutual exclusion.

If two tasks do not share variables then there is no need for mutual exclusion.

Condition synchronization is another significant requirement and is needed when a task wishes to perform an operation that can only sensibly, or safely, be performed if another task has itself taken some action or is in some defined state.

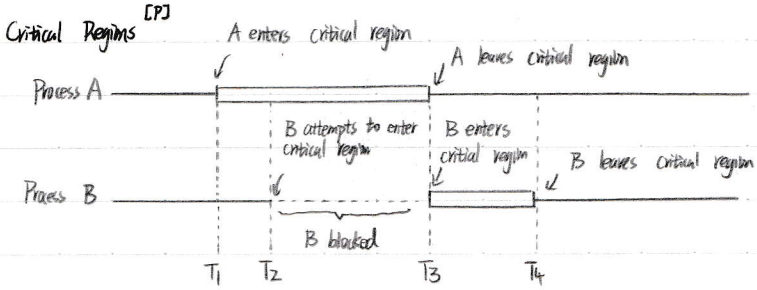
An example of condition synchronization comes with the use of buffers. Two tasks that exchange data may perform better if communication is not direct but via a buffer. This has the advantage of de-coupling the tasks and allows for small fluctuations in the speed at which the two tasks are working. For example, an input task may receive data in bursts (突发) that must be buffered for the appropriate user task. The use of a buffer to link two tasks is common in concurrent programs and is known as a producer-consumer system.

Two conditions synchronizations are necessary if a finite (bounded) buffer is used.

① The procedure producer task must not attempt to deposit data into the buffer if the buffer is full.
② The consumer task cannot be allowed to extract objects from the buffer if the buffer is empty.

Moreover, if simultaneous deposits or extractions are possible, mutual exclusion must be ensured so that two producers, for example, do not corrupt the 'next free slot' pointer of the buffer.

Critical Regions [P]



Mutual Exclusion using Critical regions (critical section)

No.

Date

Mutual Exclusion with Busy Waiting

(a) Thread 0 // (b) Thread 1

load free but beware compare with 0

```
while (true) {
    while (free==0); /*loop*/
    free = 0;
    critical_region();
    free = 1;
    noncritical_region();}
```

```
while (true) {
    while (free==0); /*loop*/ ← no choice but to loop round and recheck the flag
    free = 0;
    critical_region();
    free = 1;
    noncritical_region();}
```

[B]

One way to implement synchronization is to have tasks set and check shared variables that are acting as flags. To signal a condition, a task sets the value of a signal; to wait for this condition, another task checks this flag and proceeds only when the appropriate value is read.

Cons: { It's not possible to impose queueing disciplines easily if there is more than on task waiting on a condition (that is, checking the value of a flag)
{ They can leave to (livelock) This is an error condition where tasks get stuck in their busy-wait loops and are unable to process.

↑ This has a fatal ~~race~~ condition (~~Live Lock~~)

```
while (true) {
    while (turn != 0) /*loop*/
        critical_region();
        turn = 1;
        noncritical_region();}
```

```
while (true) {
    while (turn != 1) /*loop*/
        critical_region();
        turn = 0;
        noncritical_region();}
```

Works but strict alternation 交替

Peterson's Algorithm

```
# define FALSE 0
# define TURE (!FALSE)

volatile int turn;
volatile int interested[2];

void enter_region(int thread)
{
    int other;
    other = 1 - thread;
    interested[thread] = TRUE;
    turn = other;
    while (turn==other && interested[other]);
}
```

Volatile: 告诉编译器，在编译源码时，对变量不要使用优化

Before:

```
int *a; int b;
b = (*a)*(*a);
```

⇒

```
int *a; int b; int c;
c = *a;
b = c*c;
```

Aim: 减少存储器的读取时间

外部存储器的读写速度 < 内存读写速度

⇒ 省一次外部读取时间

在运算过程中，对变量 *a 再次读取

防止因变量 *a 的值在这一期间改变

E.g.

P0:
```
flag[0] = true;
turn = 1;
while (flag[1]==true && turn==1)
{ wait:}
flag[0] = false;
```

P1:
```
flag[1] = true
turn = 0;
while (flag[0]==true && turn==0)
加上 flag[1] = false;
```

Elaboration of the previous solution and removes the need for strict alternation

{ turn: 进入临界区的信号
{ flag: 临界区状态及哪个进程正在占用临界区

```
void leave_region(int thread)
{
    interested[thread] = FALSE;
}
```

⇒ Thread 0

```
while (true) {
    enter_region(0);
    critical_region();
    leave_region(0);
    noncritical_region();}
```

Thread 1

```
while (true) {
    enter_region(1);
    critical_region();
    leave_region(1);
    noncritical_region();}
```

This approach has two flags (turn and interested) that are manipulated by the task that owns them and a turn variable that is only used if there is {contention} for entry to the critical sections

Campus

Waste valuable Processor time ← Busy wait cons

No.

Date

TSL instruction (Test and Set Lock)

```
enter_region:
        TSL REGISTER, LOCK      ; copy lock and set lock to 1
                                ; in a single atomic operation
        CMP REGISTER, #0        ; was lock zero?
jump if not 0   JNE enter_region ; no, so try again
        RET                     ; return to a caller having claimed
region

leave_region
        MOVE LOCK, #0           ; release lock
        RET
region
```

Entering and leaving a critical region using the TSL instruction

(Unfortunately, the x86 does not have the TSL instruction)

Test and Set on the x86

XCHG Exchange Register

Logic: destination <--> source XCHG switches the contents of its operands, which may be either bytes or words

Example: LOCK XCHG SEMPHOR, DX SEMPHOR ⇄ DX

Note: Used in conjunction with the LOCK prefix, this instruction is particularly useful when implementing semaphores to control shared resources.

LOCK is a one-byte prefix that can 在...前 precede any instruction. Lock causes the processor to assert its bus logic signal while the instruction that follows is executed. If the system is configured such that the LOCK signal is used, it prevents any external device or event from accessing the bus, including interrupts and DMA transfers. (Direct Memory Access)

Note: this instruction was provided to support multiple processor systems with shared resources. In such a system, access to those resource is generally controlled via a software-hardware combination using semaphores. This instruction should only be used to prevent other bus masters from interrupting a data movement operation. This prefix should only be used with XCHG, MOV, MOVS.

[b] # Suspend and resume sleep() wait() wakeup()

Java thread class

```
public final void suspend();
public final void resume();
```

One of the problems with busy-wait loops is that they waste valuable processor time. An alternative approach is to suspend (that is, remove from the set of runnable tasks) the calling task if the condition for which it is waiting does not hold.

```
boolean Jone flag;
final boolean up = true;
final boolean down = false;

class FirstThread extends Thread {
    public void run() {
        ...
        if (flag == down) {
            suspend();
        };
        flag = down;
        ...
    }
}

class SecondThread extends Thread {
    FirstThread T1;

    public SecondThread(FirstThread T) {
        super();
        T1 = T;
    }

    public void run() {
        ...
        flag = up;
        T1.resume();
        ...
    }
}
```

Unfortunely, this approach suffers from what is called a data race condition.

A data race condition is a fault in the design of the interactions between two or more tasks whereby the result is unexpected and critically dependent on the sequence or timing of accesses to shared data.

T1 tests the flag (not suspend yet)
⇒ OS decides to preempt T1 and run T2
T2 sets the flag and resumes T1 (no effect as T1 is not suspended)
⇒ OS ....
T1 suspends itself. ⇒ RACE condition

The reason for this problem is that the flag is a shared resource which is being tested and an action is being taken which depends on its status.

No.

Date

[P]

Non-busy synchronization : Sleep and Wakeup

We can try to remove the polling load from the CPU by constructing new Operating System / runtime methods:

- sleep() will deschedule a blocked thread
- wakeup() will enable it to run again after a sleep.

```
#define N=100                          // Number of slots in the buffer
int count=0;                           // Number of items in the buffer

void producer(void)
{
    int item;

    while(true){
        item = produce_item();         // Generate next item
        if(count==N)                   // If buffer is full, go to sleep
            sleep();
        insert_item(item);             // put item in buffer
        count = count+1;               // increment count of items in buffer
        if(count==1)                   // was buffer empty?
            wakeup(consumer);
    }
}
```

```
void consumer(void)
{
    int item;

    while(true){
        if(count==0)                   // If buffer is empty
            sleep();
        item = remove_item();          // take item out of buffer
        count = count-1;
        if(count==N-1)
            wakeup(producer);
        consume_item(item);
    }
}
```

Consumer 加载 count==0，但未 sleep()，这时 producer 写入数据，count==1 并发送唤醒信号。由于 consumer 还没 sleep()，wakeup signal 丢失，consumer sleeps

Context Switching → Concurrency

The race condition can occur because access to count is unconstrained. The following situation could possibly occur. The buffer is empty and the consumer has just read count to see if it is 0. At that instant, the scheduler decides to stop running the consumer temporily and start running the procedu producer. The producer inserts an item in the buffer, increments count, and notice that it is now 1. Reasoning that count was just 0, and thus the consumer must be sleeping, the producer calls wakeup to wake the consumer up.

Unfortunely, the consumer is not yet logically asleep, so the wakeup signal is lost. When the consumer next runs, it will test the value of item it previously read, find it to be 0, and go to sleep. Sooner or later the producer will fill up the buffer and also go to sleep. Both will sleep forever.

The essence of the problem here is that a wakeup sent to a process that it not (yet) sleeping is lost. If it were not lost, everything would work. A quick fix is to modify the rules to add a wakeup waiting bit to the picture. When a wakeup is to go to sent to a process that is still awake, this bit is set. Later, when the process tries to go to sleep, if the wakeup waiting bit is on, it will be turned off, but the process will stay awake. The wakeup waiting bit is a piggy bank for wakeup signals

当两个线程访问 object 的一个 synchroniced (this) 同步代码时，~~只有一个线程在禁止访问该 or~~
一个时间内只有一个线程得到执行

No.

Date

Java's synchronized, wait() and notify()

The last example makes it clear that sleep() and wake up() will normally need to be run within a critical section
This poses another problem: the critical section will not be freed by a sleep thread
Java fixes this problem by having a set of special language primitives (基元)
synchronized marks a block or method as a critical section
wait() can only be invoked from within a synchronized unit, and releases the critical section as well as causing a sleep.
notify() can only be invoked from within a synchronized unit and issues a wakeup.

```
public final class ProducerConsumer {
    final int N = 100;
    private int count = 0;
    private Object lock = new Object();

    void producer() {
        Object item;
        try {
            while (true) {
                item = Producer.produce_item();
                synchronized (lock) {
                    while (count == N)
                        lock.wait();
                    Buffer.insert_item(item);
                    count = count + 1;
                    lock.notifyAll();
                }
            }
        }
        catch (InterruptedException e) {
            throw new AssertionError(e);
        }
    }

    void consumer() {
        Object item;
        try {
            while (true) {
                synchronized (lock) {
                    while (count == 0)
                        lock.wait();
                    item = Buffer.remove_item();
                    count = count - 1;
                    lock.notifyAll();
                }
                Comsumer.consume_item(item);
            }
        }
        catch (InterruptedException e) {
            throw new AssertionError(e);
        }
    }
}
```

wait() is declared to throw an InterruptedException but this can (should) never happen here, so we assert that it doesn't.

You must use a "while", not an "if" round the wait(); the tested condition may no longer be true when you regain the synchronized lock. Indeed, it may never have been true, the Java runtime is permitted to return you unexpected from wait().
In general, use notifyAll() to prevent deadlock: in this case, if we have a single producer and a single consumer, we can use notify()
It is safest to use a private object as the lock; if you something public, like self, there is the danger that the client might wait() it.
You should not normally allow synchronising clauses to be sub-classed: a child class can easily violate the synchronization invariants.
This whole mechanism have very bad fairness properties.

A fairness constrain is imposed on the system that it fairly select the process to be executed next.

```
Object o;

synchronized (o) {
    while (can't make progress)
        o.wait()
    do something;
        o.notifyAll();
}
```

['semәfɔ:]

Semaphores

[D]

Semaphores are a simple mechanism for programming mutual exclusion and condition synchronization.

Pros: { Simplify the protocols for synchronization
Remove the need for busy-wait loops

A semaphore is a non-negative integer variable that, apart from initialization, can only be acted upon by two procedures. These procedures are called wait and signal in this book.

wait(s) - If the value of the semaphore, S, is greater than 0 the decrement its value by 1; otherwise delay the task until S is greater than zero (and then decrement its value).

signal(s) - Increment the value of the semaphore, S, by one.

General semaphores are often called counting semaphores, as their operations increment and decrement an integer count.

The additional important property of wait and signal is that their actions are atomic (indivisible). Two tasks, both executing wait and signal operations on the same semaphore, cannot interfere with each other. Moreover, a task cannot fail during the execution of a semaphore operation.

[D]

The semaphore object has two associated operations: { down() -- or wait() or P (proberen)
up() -- or signal() or V (verhoog)

A semaphore has an associated synchronised variable.

down(): if (S>0) S--; else wait;
up(): if (Threads_waiting) wake_one; else S++;

```
task P1;
  loop
    wait(mutex);
      <critical section>
    signal(mutex);
      <non-critical>
  end
end P1
```

```
task P2;
  loop
    wait(mutex);
      <critical section>
    signal(mutex);
      <non-critical>
  end
end P2
```

```
#include <semaphore.h>

sem_t mutex, empty, full;

void initialisation(void) {
  sem_init(&mutex, 0, 1);   // Maximum # of tasks that can run simultaneously
  sem_init(&empty, 0, N);   // Third argument is initial value
  sem_init(&full, 0, 0);}
```

The producer-consumer problem using semaphores. ⇓

```
void producer(void) {
  Item *pitem;
  while(true) {
    pitem = produce_pitem();
    sem_wait(&empty);   -1
    sem_wait(&mutex);   -1
    insert_item(pitem);
    sem_post(&mutex);   +1
    sem_post(&full);}}
```

```
void consumer(void) {
  Item *pitem;
  while(true) {
    sem_wait(&full);
    sem_wait(&mutex);   -1
    pitem = remov_item();
    sem_post(&mutex);   +1
    sem_post(&empty);
    consume_pitem(pitem);}}
```

If P1 and P2 are in contention then they will execute their wait statements simultaneous. However, as wait is atomic, one task will complete execution of this statement before the other begins.

One task will execute a wait(mutex) with mutex=1, which will allow the task to proceed into its critical section and set mutex to 0; the other task will execute its wait(mutex) with mutex=0, and be delayed. Once the first task has exited its critical section, it will signal(mutex). This will cause the semaphore to become 1 again and allow the second task to enter its critical section (and set mutex to 0 again).

With a wait/signal bracket around a section of code, the initial value of the Semphore will restrict the maximum amount of concurrent execution of the code. If the initial value 0, no task will ever enter. If it is 1 then a single task may enter (that is, mutual exclusion). For values greater than one, the given number of concurrent executions of the code is allowed.

Memory management

Semaphore, and mutex and condition variable (synchronizing) functions will contain special code (memory barriers) to ensure that all caches are flushed to main memory and all threads have the same view of the state of the system. These calls are thus likely to be expensive, but they guarantee that local memory written to in on thread before a synchronization call can be read correctly from another thread after the call. So such code can share memory between threads if there is an intervening 位于中间的 synchronizing function call.

Technically, there is also the risk that the compiler might inappropriately cache local variables in registers across synchronizing function calls. So it is essential to use a POSIX-compatible compiler; C language standards before C11 do not provide the necessary guarantees.

Semaphores in Java, before Java 1.5

WARN: Bad Fairness Properties

```
public class Semaphore {
    private int value;
    public Semaphore (int initial) {
        value = initial; }
    synchronised public void up() {
        ++value;
        notify(); }
    synchronised public void down() {
        while (value==0) wait();
        --value; }
```

why? → Java's built-in concurrency constructs (synchronized, wait(), notify(), ...) do not specify which thread should be freed when a lock is released. It is up to the JVM implementation to decide which algorithm to use

Fairness gives you more control: when the lock is released, the thread with the longest wait time is given the lock (FIFO processing). Without fairness (and with a very bad algorithm) you might have a situation where a thread is always waiting for the lock because there is a continuous stream of other threads.

If the Semaphore is set to be fair, there's small overhead because it needs to maintain a sequen queue of all the threads waiting for the lock. Unless you're writing for writing a high throughput/high performance/many cores application, you probably wont see the different though!

Message Passing

The producer-consumer problem with N messages.

FreeRTOS

```
#include <FreeRTOS.h>
#include <queue.h>

xQueueHandle queue;

void initialisation (void) {
    queue = xQueueCreate (N, sizeof (Item)); }

void producer (void) {
    Item item;
    while (true) {
        item = produce_item();
        (void) xQueueSendToBack (queue, &item, portMAX_DELAY); }}

void consumer (void) {
    Item item;
    while (true) {
        (void) xQueueReceive (queue, &item, portMAX_DELAY);
        consume_item (item); }}
```

Message passing is a third approach to thread synchronization

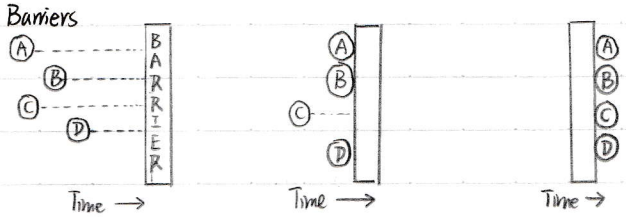
Normally have the choice of: ① Monitors and condition variables (wait, notify)
② Semaphore
③ Message passing

[Memory management]

In general it is not necessary for message-passing systems to flush caches during message passing. Some do (allowing inter-thread pointers to be used), some don't. The general assumption is that data is passed by value through the queue/mailbox.

Note that this allows alternative implications implementations. The OS could implement the queue in ordinary memory, and flush caches during message passing. It could implement the queue in special un-cached memory: avoiding the overhead of cache flushing. It could even implement the two threads on completely separate processors, and pass the queue data along a physical inter-processor communication port.

Barriers



Use of barrier:

① Processes approaching a barrier

② All processes but one blocked at barrier

③ Last process arrives, all are let through

④ Natural for "gridded" problems

No.

Date

Dining Philosopher

The dining philosophers problem is an example problem often used in concurrent algorithm design to illustrate synchronization issues and techniques for resolving them.

Problem statement

Five silent philosophers sit at round table with bolo bowls of spaghetti. Forks are placed between each pair of adjacent philosopher. Each philosopher must alternatively think and eat. However, a philosopher can only eat spaghetti when they have both left and right forks. Each fork can be hold by only one philosopher and a philosopher can use the fork only if it is not being used by another philosopher. After an individual philosopher finishes eating, they need to put down both forks so that the forks became available to others.

The problem is how to design a discipline of behavior (a concurrent algorithm) such that no philosopher will ~~starve~~ starve (餓死)

Problems

The problem was designed to illustrate the challenges of avoiding deadlock, a system state in which no progress is possible.

Consider a proposal in which each philosopher is constructed to behave as follows:

1. think until the left fork is available; when it is, pick it up.
2. think until the right fork is available; when it is, pick it up.
3. when both forks are held, eat for a fixed time (amount of)
4. then put the right fork down
5. then put the left fork down
6. repeat from the beginning

This attempted solution fails because it allows the system to reach a dead lock, in which no progress is possible. This is a state in which each philosopher has picked up the fork to the left, and is waiting for the fork to the right to become available, vice versa. a fork

With the given instructions, this state can be reached, and when it is reached, the philosophers will eternally (永遠に) wait for each other to release

Resource starvation might also occur independently of deadlock if a particular philosopher is unable to acquire both forks because of a timing problem. For example, there might be a rule that the philosophers put down a fork after waiting ten minutes for the other fork to become available and wait a further ten minutes before making their next attempt. This scheme eliminates the possibility of deadlock. but still suffers from the problem of livelock. If all philosophers appear in the dining room at exactly the same time and each picks up the left fork at the same time the philosopher will wait ten minutes until they all put their forks down and then wait a further ten minutes before they all pick them up again.

Solution

[Resource hierarchy solution]

Assign partial order to the resources (the forks), and establishes the conversion that all resources will be requested in order, and that no two resources unrelated by order will ever be used by a single unit to work. Here the resources (forks) will be number 1 through 5 and each unit of work (philosopher) will always pick up the lower-number fork first, and then higher-number fork, from among the two forks they plan to use.

In this case, if four of the five philosophers pick their lowered-numbered fork, only the highest-numbered fork will remain on the table, so the fifth philosopher will not be able to pick up any fork.

[Arbitrator (仲裁人) solution]

Guarantee that a philosopher can only pick up both forks or none by introducing an arbitrator, e.g., a waiter. In order to pick up the forks, a philosopher must ask permission of the waiter. The waiter gives permission to only one philosopher at a time until the philosopher picked up both of their forks. Putting down the fork is always allowed.

The waiter can be implemented as a mutex. This approach can result in reduced parallism

No.

Date

[Cundy / Misra solution]

Allow for arbitrary agents to contend for an arbitrary number of resources (However, it violates the requirement that "the philosophers do not speak to each other.)

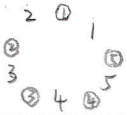
1. For every pair of philosophers contending for a resource, create a fork and give it to the philosopher with the lower ID (n for agent P_n). Each fork can either be dirty or clean. Initially, all forks are dirty.

2. When a philosopher wants to use a set of resources (i.e. eat), said philosopher must obtain the forks from their contending ~~nig~~ neighbours. For all such forks the philosopher does not have, they send a request message.

3. When a philosopher with a fork receives a request message, they keep the fork if it is clean, but give it up when it is dirty. If the philosopher sends the fork over, they clean the fork before doing so.

4. After a philosopher is done eating, all their forks become dirty. If another philosopher had previously requested one of the forks, the philosopher that has just finished eating cleans the fork and sends it.

This solution allows for a large degree of concurrency, and will solve an arbitrarily large problem.



No.

Date

[P]

Dining Philosopher - Java

Modify the run() method so that philosopher zero takes her forks in the opposite sequence to the other philosophers
⇒ This breaks the dependency cycle and ensures that the code is live.
It does NOT provide a guarantee of fairness

A Non-solution to the dining philosophers problem

A philosopher is modeled as an instance of the class DiningPhilosophers running as a thread.

A fork is a "passive" instance of the class Fork.

```
import java.io.*;

class DiningPhilosophers extends Thread {
    static final int N=2;
    private static Fork[] forks = new Fork[N];
    private int number;

    DiningPhilosophers(int number){
        System.out.println("Construct philosopher " + number);
        this.number = number;}

    public void run(){
        System.out.println("Run philosopher " + number);
        while(true){
            System.out.println("Philosopher " + number + " thinks");
            doze(50);
            forks[number].take();
            doze(5);
            forks[(number+1)%N].take();
            System.out.println("Philosopher " + number + " eats");
            doze(50);
            forks[number].drop();
            forks[(number+1)%N].drop();}}

    public int getNumber(){
        return number;}

    public void doze(int msecs){
        try{
            Thread.sleep(msecs);}
        catch{InterruptException e){
            throw new AssertionError(e);}}}

    public static final void main(String[]s){
        for(int i=0;i<N;i++)
            forks[i] = new Fork(i);
        for(int i=0;i<N;i++)
            new DiningPhilosophers(i).start();}}
```

if number == 0 ⇐ 拿(num+1)%N ⇒ avoid deadlock

The code appears to run

BUT each fork is modeled as a simple object, there is nothing to stop a single fork being held by two philosophers at once. The code is unsafe.

```
import java.util.concurrent.*;
class Fork{
    Semaphore s;
    int number;
    Fork(int number){
        this.number = number;
        s= new Semaphore(1);}

    void take(){
        DiningPhilosophers p = ....
        System....
        ⇒ s.acquireUninterruptibly();
        ...}

    void drop(){
        ...
        ⇒ s.release();}}
```

Modify the definition of the Fork class so to include a binary Semaphore. This means ensures that a given fork can be taken by at most one philosopher.

拿(num) 拿(num+1) The code is now safe but suffers from possible deadlock as a cycle of dependency can arise after each philosopher has taken one fork

```
class Fork{
    int number;
    Fork(int number){
        this.number = number;
        ~~System.out.println("Philosopher " + P.getNum() + " takes fork " + number);~~
        System.out.println("Construct fork " + number);}

    void take(){
        DiningPhilosophers p= (DiningPhilosophers) Thread.currentThread();
        System.out.println("Philosopher " + P.getNumber() + " takes fork " + number);}

    void drop(){
        DiningPhilosophers p = (DiningPhilosophers) Thread.currentThread();
        System.out.println(...);}}
```

Modification based on hierarchy solution

The Readers and Writers Problem

POSIX C

```
#include <semaphore.h>

sem_t mutex, db;
int rc=0;

void initialization(void){
    sem_init(&mutex,0,1);
    sem_init(&db,0,1);
    rc=0;}

void reader(void){
    while(true){
        sem_wait(&mutex);
        rc=rc+1;
        if(rc==1)
            sem_wait(&db);
        sem_post(&mutex);
        read_database();
        sem_wait(&mutex);
        rc=rc-1;
        if(rc==0)
            sem_post(&db);
        sem_post(&mutex);
        use_data();}}

void writer(void){
    while(true){
        invent_data();
        sem_wait(&db);
        write_database();
        sem_post(&db);}}
```

The classic synchronization problem. Many readers can access the database at the same time, but only one writer, which must have exclusive access.

This solution gives readers priority; a writer might never gain access.

Atomic

Old style

```
public final class Counter {
    private long value = 0;
    public synchronized long getValue() {
        return value;
    }

    public synchronized long increment() {
        return ++value;
    }
}
```

New style

```
import java.util.concurrent.atomic.AtomicLong;

public final class NonblockingCounter {
    private AtomicLong value;

    public long getValue() {
        return value.get();
    }
    public long increment() {
        int v;
        do {
            v = value.get();
        } while (!value.compareAndSet(v, v+1));
        return v+1;
    }
}
```

如果设置没成功（当前值无效，即被其他线程操改）

OR

```
int current = get();
int next = current + 1;
if (compareAndSet(current, next))
    return next;
```

```
static AtomicInteger i;
...
int current = i.incrementAndGet();
```

is a Thread-Safe version of ⇒

```
static int i;
...
int current = ++i;
```

maps:

- ++i → ~~i.incrementAndGet~~ i.incrementAndGet()
- i++ → i.getAndIncrement()
- --i → i.decrementAndGet()
- i-- → i.getAndDecrement()
- i=X → i.set(x)
- X=i → X=i.get(X)

[ABA problem]

In ~~multiprocess~~ multithread computing, the ABA problem occurs during synchronization, when a location is read twice, has the same value for both reads, and "value is the same" is used to indicate "nothing has changed". However, another thread can execute between the two reads and change the value, do other work, then change the value back, thus fooling the first thread into thinking "nothing has changed" even though the second thread did work that violates that assumption

The ABA problem occurs when multiple threads (or processes) accessing shared data interleave.

① Process P1 reads value A from shared memory [Hiden Modification in shared memory].
② P1 is preempted, allowing P2 to run
③ P2 modifies the shared memory value A to value B and back to A before preemption.
④ P1 begins execution again, see that the shared memory value has not changed and continues.

Java虚拟机

The JVM has the opportunity to implement this using LOCK CMPXCHG r/m32, r32. (compare and exchange)

↑ Compare EAX with r/m32.
If equal, ZF is set and r32 is loaded into r/m32
Else clear ZF and load r/m32 into EAX
EAX is the accumulator register as a 32-bit value

No.

Date

C++ 11 threads

[Simple Example]

```
#include <iostream>
#include <thread>

void call_from_thread() {
    std::cout << "Hello, ELEC1204 " << std::endl;
}
int main() {
    std::thread t1 (call_from_thread);  // Launch a thread

    t1.join();                          // Join the thread with the main thread.
    return 0;
}
```

[Start Multiple threads]

```
#include <iostream>
#include <thread>

void call_from_thread2 (char c, int i) {
    while (i--) {
        std::cout << c << std::flush;   // non-buffered: printing happens immediately
    }
    std::cout << std::endl;
}

static const char letters[] = "ABCDEFGHIJ";

int main() {
    std::thread threads[10];
    for (int i=0; i<10; i++) {
        threads[i] = std::thread (call_from_thread2, letters[i], i+10);
    }
    for (int i=0; i<10; i++) {
        threads[i].join();
    }
    return 0;
}
```

Output :

```
EEEEEEEEEE GGGGGGGGGG
A CCCCCCCCC
HHHHHHHHHH
DDDDDDDDD
IIIIIIIIII FFFFFFFFF
AAAAAAAAA
BBBBBBBBBB

EEEE
JJJJJJJJJJ
```

The threads run in no particular order. Unfortunately, they do not individually run into completion, so the output from the various threads is all mixed up.

In order to fix this problem, we need to use a mutex, also known as a critical section, or (in Java) a synchronized block.

- std = c++11

```
# include <iostream>
# include <thread>

using namespace std;

mutex m;

void call_from_thread2 (char c, int i){
  -> m.lock();
    while (i--){
        cout << c << flush;
    }
    cout << endl;
  -> m.unlock();
}
```

Still no guaranteed order of execution of the threads, but each runs to completion before the next.

11001 -7
+01001 9
10010

Lock Guards and Unique Locks

```
void call_from_thread2 (char c, int i){
    std::lock_guard <std::mutex> ul (m);
    while (i--){
        std::cout << c << std::flush;
    }
    std::out << std::endl;
}   //exiting the block releases the mutex when
                                ul is destoryed
```

[免责]

The lock_guard is a wrapper for a mutex. The mutex is automatically claimed when the lock_guard is constructed and is released when it is destructed; If it is an automatic variable, this happens when the block in which it is declared exist exits.

The way of locking the mutex also ensures it will be released if the block exists because an exception is thrown. That may or may not be a good idea. What do you want your code to do if an exception is thrown while holding a mutex.

① Release the lock, this allowing other threads to access the shared data in a possibly inconsistent state, or

② keep the lock, preventing erroneous behaviour else where, but pontentially causing deadlock

The unique_lock is a version of lock_guard that should be used with condition variable.

Condition Variable.

A simple mutex is not enough to allow a thread to wait until a resource becomes available. For example, if our thread is trying to input data from a buffer, it must wait while the buffer is empty until another thread puts some data into the buffer. We need to use a mutex to protect the shared variable, such as the buffer itself, but we also need to use a condition variable as a place to wait, with the mutex released, until more data arrives.

```
# include <iostream>
# include <thread>

char buffer;
int buffer_content = 0;
std::mutex m;
std::condition_variable cv;

void call_from_thread() {
    while (true) {
        std::unique_lock <std::mutex> ul1(m);
        while (buffer_contents == 0) {
            cv.wait (ul1);
        }
        std::cout << buffer << std::flush;
        buffer_contents --;
        cv.notify_all();
    }
}
//exiting the block releases the mutex
  when ul1 is destoryed
```

```
int main() {
    std::thread t1 (call_from_thread);

    while (true) {
        std::unique_lock <std::mutex> ul2 (m);
    }
    while (buffer_contents == 1) {
        cv.wait (ul2);
    }
    std::cin >> buffer;
    buffer_contents++;
    cv.notifyall();
    }
}
```

lock
cv 是在満足 No → wait
Yes
↓
cv++
cv.notify()

Mutex condition variable pattern

There is a general pattern that provides safe multithreading of resources.

```
std:: mutex m;
std:: condition_variable cv;

one_of_the_thread_function(...){
{
    std:: unique_lock <std:: mutex> lk(m);
    while (... cannot make progress ...){
        cv.wait(lk);
    }
    ... do what you have to do ...
    cv.notify_all(); // wake up all waiting threads
} // exiting the block releases the mutex when lk is destroyed.
```

This particular pattern is pretty foolproof. Every thread trying to modify the shared data first acquires a lock on the mutex to ensure safe sharing. It is then free to explore (read) the shared data to decide whether there is a resource available which allows it to make progress. If not, it waits, releasing the mutex as it does so. If it can progress, it does. When it makes progress, it wakes up all waiting threads, thus ensuring no possibility of deadlock. The woken threads take turns to again check resources (the while() loop) and, if they can, they make progress of their own.

No.

Date

Model Checking

Verifying the correctness of our code

Programming is not like algebra
- Variables are not unknowns. They change with time.

Testing is limited value
- If you know all the answers, replace the program with a look-up table
- Does not test for (further) compiler optimisations
- Does not test for multi-threaded non-determinism

We want to verify against a specification.

What can a program do?

① Break something (e.g. burn out the hardware)
② Fail to complete
③ Complete too late
④ Complete and give the wrong result.
⑤ Complete and give the right result

Assertions

The simplest specification is a (point) assertion. It is supported by the C standard

```
int mul (const int ain, const int b) {
    int a = ain;
    int result = 0;                    when result overflows
    while (a--) {                      // It still passes the assertion
        result = result + b;}
    assert (result == ain*b);
    return result;
}
```

What can go wrong?

Firstly, assume the C int is a (mathematical (unbounded) integer.
- Then the program is partially correct. If it terminates, it terminates with the right answer.

But a genuine C int is bounded by INT_MAX and INT_MIN.
- So the multiplication and the addition might overflow.
- C does not specify what happens if they do.
- Java guarantees to give the wrong answer.

We fix this with non-standard assume().

assert & assume

Assert is something that you expect the static checker be able to prove at compile time, whereas Assume is something the static check can rely upon

```
assert (const int pred) {
    if (!pred) {
        exit (EXIT_FAILURE);}}
```

Assert is a claim about the program's behaviour; It fails if the predicate is false. Asserts from the program's specification.

```
assume (const int pred) {
    if (!pred) {
        exit (EXIT_SUCCESS);}}
```

Assume is a way of restricting the DOMAIN of a function. We don't care how the program behaves when the assume is false

Corrected version of program

```
int mul (const int ain, const int b) {
    int a = ain;
    assume (a>=0);
    assume ((a==0) || ((b<=(MAX_INT/a)) && (b>=(MIN_INT/a))));

    int result = 0;
    while (a--) {
        result = result + b;
    }

    ~~assert (result + b~~
    assert (result == ain*b);

    return result;
}
```

For bounded model checking, need to unroll the loop

```
while (a--) {result = result + b;}
```

becomes, if we bound the unrolling at nine iterations

```
if(a) {a = a-1; result = result + b;}
if(a) {a = a-1; result = result + b;}
...
...
...
...
...
...
if(a) {a = a-1; result = result + b;}
if(a) assert (false); // Unrolling assertion
```

No.

Date

In bounded model checking, we can only cope with loops up to a bounded depth.

The unwinding assertion checks that the depth is never exceeded. In this case, it will fail.

We can also make the unwinding assumption that the depth is not exceeded. This prunes all input values that would have needed more depth. So it is, in general, unsafe.

We need to replace the variables with single-assignment unknowns.

The unrolled program becomes:

```
int mul(const int a, const int b){
    int a_0 = a;

    assume(a_0>=0);
    assume((a==0)||(b<=(MAX_INT/a))&&(b>=(MIN_INT/a)));

    int result_0=0;
          当a在为0时，保持originally
    int a_1 = a_0 ? a_0-1 : a_0; int result_1 = a_0 ? result_0+b : result_0;
    int a_2 = a_1 ? a_1-1 : a_1; int result_2 = a_1 ? result_1+b : result_1;
    ...
    ...
    ...
    int a_9 = a_8 ? a_8-1 : a_8; int result_9 = a_8 ? result_8+b : result_8;

    assert(!a_9); 如果a_9为正数，the depth is never exceeded, in this case, it will fail.
    assert(result_9==a*b);
    return result_9;
}
```

n≤9都可以检测

We now have ordinary algebra

We look for satisfying assignments to the integer unknowns which would cause the assertion to fail. The program is broken if there is a set of assignments to a, b, a_0 ... a_9, result_0 ... result_9 which makes the following expression true:

```
(a_0 == a)&
(a_0>=0)&
((a==0)||((b<=(MAX_INT/a))&(b>=MAX_INT/a)))&
(result_0==0)&
(a_1==a_0 ? a_0-1 : a_0)&(result_1==a_0 ? result_0+b : result_0)&
(a_2==a_1 ? a_1-1 : a_1)&(result_2==a_1 ? result_1+b : result_1)&
...
...
(a_9==a_8 ? a_8-1 : a_8)&(result_9==a_8 ? result_8+b : result_8)&
((a_9!=0)|(result_9 != a*b));
```

1+2+3+...+n

No.

Date

Proof by induction (帰納法)

```
unsigned int sumint (const unsigned int n) {
    unsigned int i = 0;
    unsigned int result = 0;
    while (i<n) {
        i = i+1;
        result = result + i;
    }
    assert (result == n*(n+1)/2);
    return result;
}
```

① ②

1. Assert an equality claimed to be valid for all n
2. Show that it is true for n=0
3. Show that, if it is true for n=m, then it is also true for n=m+1
4. Then
   - the n=0 case implies then n=1 case
   - the n=1 case implies then n=2 case
   - the n=2 case implies then n=3 case
   - the n=3 case implies then n=4 case
   - and so on for all n>=0

The invariant

For a proof by induction, we need an inductive hypothesis, inside the loop

① assert (result == i*(i+1)/2);
② assert (i<=n);

Not enough, we also need to control the induction variable.

Whence come these invariants?

In general, there is no way to deduce (演繹的) them
- halting problem (the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running or continute to run forever).

For simple loops, they can sometimes be guessed.
Range analysis will often control the induction variable.
- We know (i<n) at the top of the lap
- We know that i is incremented by exactly one, so (i-1<n) at the bottom of the loop.

Now the proof falls into three parts

[Base case]

```
unsigned int i = 0;
unsigned int result = 0;
if (i<n) {
    i = i+1;
    result = result + i;
    assert (result == i*(i+1)/2;
    assert (i<=n);
}
else {
    assert (result == n*(n+1)/2);
}
```

The base case can be extended to perform some bounded unrolling as well

[Inductive step]

```
unsigned int i;
unsigned int result;
assume (result = i*(i+1)/2);
assume (i<=n);
if (i<n) {
    i = i+1;
    result = result + i;
    assert (result == i*(i+1)/2);
    assert (i<=n);
}
```

Sometimes it helps to unroll the inductive step a few times (k-induction)

[Termination Condition]

```
unsigned int i;
unsigned int result;
assume (result == i*(i+1)/2);
assume (i<=n);
if (!(i<n)) {
    assert (result == n*(n+1)/2);
}
```

The inductive step and the termination condition can be fused into a single test.

x(x+6) → x²+6x

It is all much harder in practice

Think about arrays and pointers
For OO languages, worry about dynamic dispatch (method pointers)
Concurrency is a real pain; all valid interleavings must be considered
Our tool is ESBMC //esbmc.org/
Our (Mikhail's) latest achievement is the incorporation of a theory of IEEE floating ~~number~~ point numbers.

KOKUYO

No.

Date

Model Checking Synchronisation

What is a model checker

Model checkers investigate the behaviour of a (possibly nondeterministic) finite state machine.

The machine is typically built up as a composition of smaller finite state machines which interact through shared events

The FSM can be set up as a model of logic hardware, a computer program, a communications protocol.

Nondeterminism in the model checking context is quite different from the angelic nondeterminism finite automata magically moved to the correct next state if the current state had null (epsilon) transitions or multiple transitions on the current symbol.

Our nondeterministic state machines may pick any available outgoing transition, or any null (tau) transition. It is up to the model checker to find out if anything bad might happen.

What a model checker does?

A model checker explores all possible states of a nondeterministic system.

This differs from a simulator & in that all possible futures are explored. when there is some ambiguity (nondeterminism) about the next state, & a simulator makes a choice; a model checker explores all possible choices

For model checkering to be possible, the system must have a finite number of possible states.

Model checking can be very slow; slowness ~~exponential~~ exponential in the number of states is possible.

A simple example

The following example, a simple mutex, captures some of the essential ideas in model checking. It does not capture the richness of the notation (even in LTSA) nor does it discuss the use of temporal logic or of process algebras (such as CSP and CCS) to capture properties and specifications of systems.

A crude mutex

Let us to achieve mutual exclusion between two threads by defining a turn variable.

```
int turn = 1;
```

and in each thread (me = 0, 1 respectively) we guard the critical section as follows:

```
while (turn != me); /* loop waiting for my turn */
/* Critical section */
turn = 1 - me;
```

Is the mutex too crude?

We are going to use LTSA LTSA (Labelled Transition System Analyser) to find out if this mutex does what it should.

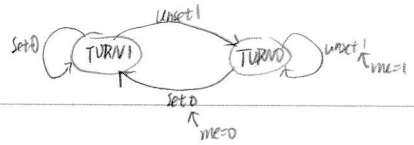
The key questions are: Does it maintain mutual exclusion and avoid corrupting data?

Does it avoid deadlock?

turn = 1 - me

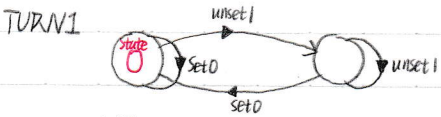
No.

Date



Modelling the mutex

First, we construct a model of the variable turn. A variable is always willing to engage in any event which changes it, so our variable behaves as the following state machine.



Note that the states of TURN1 are numbered by LTSA from the start state. So state 0 corresponds to turn=1 and state 1 to turn=0

The turn variable

Events of the same name are shared by all processes (machines) that have them in their vocabulary. The two threads interact separately with the variable, so we need to distinguish the actions of the two threads with suffices 0 and 1.
We do not need set 1 and ~~set0~~ unset 0 event as they are not used by the threads.
The variable may be represented in FSP by the definition:

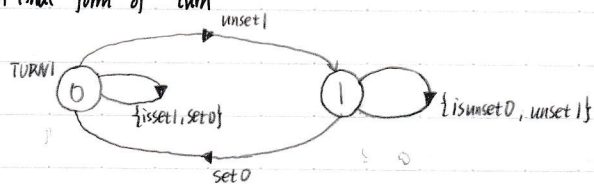
```
TURN1 = (set 0 -> TURN1 | unset 1 -> TURN0)
TURN0 = (set 0 -> TURN1 | unset 1 -> TURN0)
```

Testing the variable

A simple model for testing a variable is to introduce an event in which the variable will only engage if it has the desired value. Then the testing thread will be locked when it tries to engage in the testing event when the variable has the wrong value. The testing event does not, of course, change the value of the variable.

```
TURN1 = (set 0 -> TURN1 | unset 1 -> TURN 0 | isset 1 -> TURN1)
TURN0 = (set 0 -> TURN1 | unset 1 -> TURN 0 | isunset 0 -> TURN0)
```

Final form of turn



P0 isunset0 → TURN0 (State 1)

P1 isset 1 → TURN1 (State 0)

Recap: variables

A variable has a state for each possible value.

For each value to which each thread can set the variable there is a named event. There is a transition on this event from every value of the variable to the new value.

For each thread and for each value that is tested by it, there is a named event. There is a transition on this event from the values at which the test is true back to themselves.

FSP rules

First, note that we use tail recursion (尾递归) to define our processes. That is, a process is typically defined to engage in some event, then behaves as another named process.

An event can fire if every process named in the composite system which has the event in its vocabulary is able to engage in the event from its current state.

The model checker explores all possible ordering of event firings.

An event is in a process's vocabulary if there is some state in which the process will engage in the event.

It is possible, but not very useful, to add an event to a process' vocabulary explicitly; the event will never happen if it wasn't already in the vocabulary, as there is no actual state of the process which can engage in it.

Modelling one of the threads

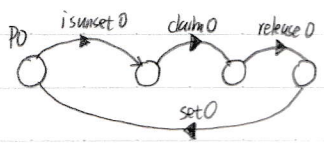
We can model the wait loop simply with a transition on the appropriate is ... event. Note that this choice replaces a busy (polling) wait with a non-busy wait. This may or may not matter, depending on the implementation.

We also add claim and release events so we can follow what is happening to the critical section:

P0 = (isunset0 → claim0 → release0 → set0 → P0).

The thread model

We remember to set turn after releasing the critical section, and get the very simple state machine:



Our complete system

TURN1 = (set0 → TURN1 | unset1 → TURN0 | isset1 → TURN1)
TURN0 = (set0 → TURN1 | unset1 → TURN0 | isunset0 → TURN0)
P0 = (isunset0 → claim0 → release0 → set0 → P0)
P1 = (isset1 → claim1 → release1 → unset1 → P1).

We can type this into LTSA and minimise

What we have achieved

We have built the model system
We have confirmed it does not deadlock of its own accord
We have not checked its safety, that is that it maintains mutual exclusion
we have not checked that it does not deadlock in its intended environment

No.

Date

The safety property

We need to ~~ent~~ ensure that only one thread holds the mutex at a time. The permitted sequence of events is:

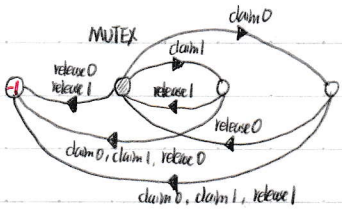
MUTEX = (claim0 → release0 → MUTEX | claim1 → release1 → MUTEX)

The important idea is that, if our mutex allows any other behaviours, then an error should be reported. That is, apart from the transitions listed above, the process should be filled out with transitions to an error state in all other events in its vocabulary.

The property notation

FSP provides a shorthand for these processes with added error (-1) state, the property keyword.

property MUTEX = (claim0 → release0 → MUTEX | claim1 → release1 → MUTEX)



The full safety model

TURN1 = (set0 → TURN1 | unset1 → TURN0 | isset1 → TURN1)
TURN0 = (set0 → TURN1 | unset1 → TURN0 | isunset0 → TURN0)
P0 = (isunset0 → claim0 → release0 → set0 → P0)
P1 = (isset1 → claim1 → release1 → unset1 → P1)
property MUTEX (claim0 → release0 → MUTEX | claim1 → release1 → MUTEX) || SYS = (TURN1 || P0 || P1 || MUTEX)

AND again = No deadlocks/errors. So far so good; our system is a mutex.

Liveness

We have to check liveness. Our mutex should be willing to accept threads trying to claim it in any order. Thus it should be avoid deadlock in a configuration where the external environment choose the order in which threads acquire the mutex.
In FSP, this is achieved by having an environment which makes a hidden internal choice into a state which offers claim0 or sta which offers claim1.

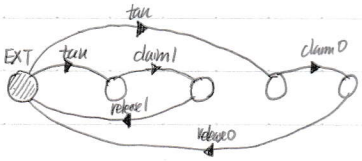
The liveness test

EXT = (t → claim0 → release0 → EXT | t → claim1 → release1 → EXT) \ {t}

This introduces a new notation. The internal choice arises because there are two different initial transition on event t. The t event is then hidden with the \ operator; the overall effect is that the process makes an internal choice as to whether to engage in claim0 or claim1

Tau events

In the graph of EXT, the hidden t event becomes a tau event because it is hidden



Campus

Final liveness test system

TURN1 = (set0 → TURN1 | unset1 → TURN0 | isset1 → TURN1),
TURN0 = (set0 → TURN1 | unset1 → TURN01 isunset0 → TURN0),

P0 = (isunset0 → claim0 → release0 → set0 → P0).
P1 = (isset1 → claim1 → release1 → unset1 → P1).
EXT = (t → claim0 → release0 → EXT | t → claim1 → release1 → EXT) \ {t}.
||SYS = (TURN1 || P0 || P1 || EXT).

LTS Analyser:

Trace to Deadlock: tau
isset1

This test fails because the mutex imposes strict alternation.
To do better, we need Peterson's algorithm

It was shown to fail to avoid deadlock when presented with requests in an externally imposed order. This is because the simple mutex imposes strict alternation.

isunset0 → TURN0
isset1 → TURN1

No.

Date

[Web]

A sequential program has a single thread of control.

A concurrent program has multiple threads of control allowing it perform multiple computations in parallel and to control multiple external activities which occur at the same time.

Concurrency is widespread but error prone

process 1: x:=x+1 (x shared variable)
process 2: x:=x-1

Assuming read and write are atomic, the result depends on the order of read and write operations on x (Race condition)

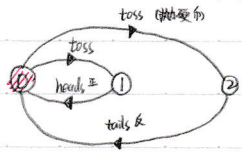
There are 4 atomic x-operations:
Process 1 reads x (R1), writes to x (W1) — R1 must happen before W1 and R2 before W2
Process 2 reads x (R2), writes to x (W2) — Thus these operation can be sequenced in 6 ways (x initially 0)

R1	R1	R1	R2	R2	R2
W1	R2	R2	R1	R1	W2
R2	W1	W2	W1	W2	R1
W2	W2	W1	W2	W1	W1
0	-1	1	-1	1	0

Final value of x is -1, 0, or 1 ⇒ Thus, non-deterministic
The result can vary from execution to execution

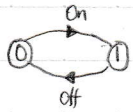
Models and Model Checking

A model is an abstract, simplified representation of the real world.
Models are described using state machines, known as Labelled Transition Systems (LTS) These are textually described in a Process Algebra as finite state processes (FSP) and displayed and analysed by the LTSA model checking analysis tool.



COIN = (toss → HEADS | toss → TAILS),
HEADS = (heads → COIN),
TAILS = (tails → COIN).

Processes and Threads



a light switch LTS
on → off → on → off →... a sequence of actions or trace

[Modelling processes]

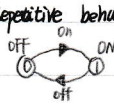
A process is the execution of a sequential program. It is modelled as a finite state machine which transits from state to state by executing a sequence of atomic actions.

If x is an action and P a process then (x→P) describes a process that initially engages in the action x and then behaves exactly as described by P

ONESHOT = (once → STOP)
← ONESHOT state machine (terminating process)

○ Convention: action begin with lowercase letters
Process begin with uppercase letters

⇒ Repetitive behaviour uses recursion:



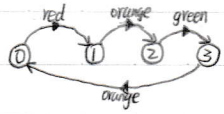
SWITCH = OFF,
OFF = (on → ON),
ON = (off → OFF).

⇒ Substituting to get a more succinct definition:
SWITCH = OFF
OFF = (on → (off → OFF)).

⇒ And again
SWITCH = (on → off → SWITCH).

Scope: OFF and ON are local subprocess definitions, local to the SWITCH definition.

⇒ FSP model of a traffic light:
TRAFFICLIGHT = (red → orange → green → orange → TRAFFICLIGHT
Trace: red → orange → green → orange → red → orange → gree ...

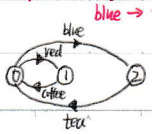


LTS generated using LTSA

If x and y are actions then (x→P | y→Q) describes a process which initially engages in either of the actions x or y. After the first action has occurred, the subsequent behaviour is described by P if the first action was x and Q is the first action is y

⇒ FSP model of a drinks machine
DRINKS = (red → coffee → DRINKS |
blue → tea → DRINKS).

LTS:



Process (x→P | x→Q) describes a process which engages in x and then behaves as either P or Q).
⇒ Tossing a coin
COIN = (toss → HEADS | toss → TAILS),
HEADS = (heads → COIN)
TAILS = (tails → COIN)

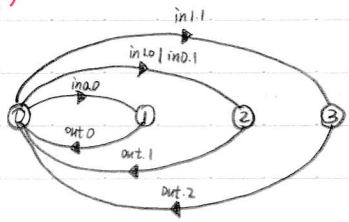
↖ 左上图

⇒ Single slot buffer that inputs a value in the range 0 to 3 and then outputs that value

BUFF = (in[i:0...3] → out[i] → BUFF) equivalent to BUFF = (in[0] → out[0] → BUFF | in[1] → out[1] → BUFF | in[2] → out[2] → BUFF | in[3] → out[3] → BUFF).

or using a process parameter with default value:

BUFF(N=3) = (in[i:0..N] → out[i] → BUFF)

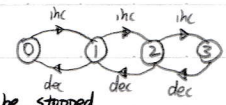


⇒ index expressions to model calculation:

const N=1
range T=0..N
range R=0..2*N
SUM = (in[a:T][b:T] → TOTAL[a+b]),
TOTAL[s:R] = (out[s] → SUM).

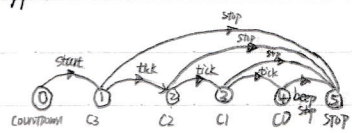
The choice (when B x→P | y→Q) means that when the guard B is true then the action x and y are both eligible to be chosen otherwise if B is false then the action x cannot be chosen.

⇒ COUNT(N=3) = COUNT[0]
COUNT[i:0..N] = (when(i<N) inc → COUNT[i+1] | when(i>0) dec → COUNT[i-1]).



⇒ A countdown timer which beeps after N ticks, or can be stopped

COUNTDOWN(N=3) = (start → COUNTDOWN[N]),
COUNTDOWN[i:0..N] = (when(i>0) tick → COUNTDOWN[i-1] | ~~when(i<N) tick~~ when(i==0) beep → STOP | stop → STOP).



Concurrent Execution

{ (P||Q) = (Q||P) - Commutative (交換)
{ (P||(Q||R)) = ((P||Q)||R) = (P||Q||R) - Associative (結合)

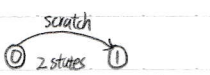
If P and Q are processes then (P||Q) represents the concurrent execution of P and Q. The operator || is the parallel composition operator.

⇒ ITCH = (scratch → STOP).
CONVERSE = (think → talk → STOP).
||CONVERSE_ITCH = (ITCH || CONVERSE).

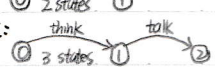
Disjoint 互斥

think → talk → scratch → think → scratch → talk → scratch → think → talk
↳ Possible traces as a result of action interleaving.

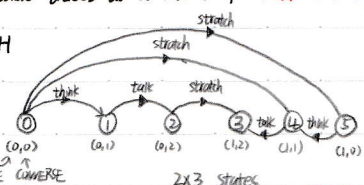
ITCH:



CONVERSE:



CONVERSE_ITCH

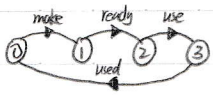


think → talk → scratch
think → scratch → talk
scratch → think → talk
scratch → talk → think

If processes in a composition have actions in common, these actions are said to be shared. Shared actions are the way that process interaction is modeled. While unshared actions may be arbitrarily interleaved, a shared action must be executed at the same time by all praeses that participate in the shared action.

⇒ A handshake is an action acknowledged by another:

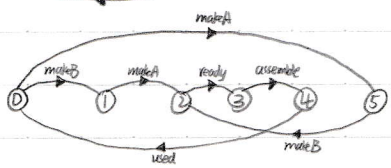
MAKER_v2 = (make → ready → used → MAKER_v2).
USER_v2 = (ready → use → used → USER_v2).
||MAKER_v2_USER_v2 = (MAKER_v2 || USER_v2).



3 states × 3 states ?
→ 4 states
Interaction constrains the overall behaviour

⇒ Multi-party synchronization

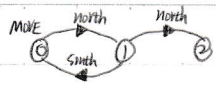
~~MAKER_V2~~ MAKER_A = (make.A → ready → used → MAKER_A).
MAKER_B = (make.B → ready → used → MAKER_B).
ASSEMBLE = (ready → assemble → used → ASSEMBLE).
||FACTORY = (MAKE_A || MAKE_B || ASSEMBLE).



No.

Date

deadlock



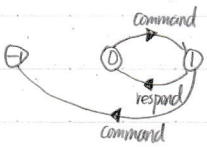
Deadlock state is one with no outgoing transitions

MOVE = (north → (south → MOVE | north → STOP)).

Safety

A safety property asserts that nothing bad happens
(① STOP or deadlock state (no outgoing transitions)
② ERROR process (-1) to detect erroneous behaviour)

E.g. ACTUATOR = (command → ACTION),
ACTION = (respond → ACTUATOR | command → ERROR).



Analysis using LTSA: Trace to ERROR: command
command

In complex systems, it is usually better to specify safety properties by stating directly what is required.

property SAFE_ACTUATOR = (command → respond → SAFE_ACTUATOR).

Liveness

A safety asserts that nothing bad happens
A liveness property asserts that something good eventually happens.

Fair choice: If a choice over a set of transitions is executed infinitely often, then every transition in the set will be executed infinitely often

No.

Date

Multithread Programming (POSIX pthreads Tutorial)

Mutual Exclusion

Mutual exclusion is a method of serializing access to shared resources. You don't want a thread to be modifying a variable that is already in the process of being modified by another thread! Another scenario is the dirty read (中途读取) where the value is in the process of being updated and another thread reads an old value.

Mutual exclusion allows a programmer to create a defined protocol for serializing access to shared data or resources. Logically, a mutex is a lock that one can virtually attach to some resource. If a thread wishes to modify or read a value from a shared resource, the thread must first gain the lock. Once it has the lock it may do what it wants with the shared value resource without concerns of other threads accessing the shared resource because other threads will have to wait. Once the thread finishes using the shared variable resource, it unlocks the mutex, which allows other threads to access the resource. This is a protocol that serializes access to the shared resource. Note that such a protocol must be enforced for the data or resource a mutex is protecting across all threads that may touch the resource being protected. If the protol is violated (e.g. If the thread modifies a shared resource without first requesting a mutex lock), the the protocol defined by the programmer has failed. There is nothing preventing a thread programmer, whether unintentionally or intentionally from implementing a flawed (有缺陷的) serialization protocol.

As an analogy, you can think of a mutex as a safe with only one key (for a standard mutex case), and the resource it is protecting lies within the safe. Only one person can have the key to the chest at any time, therefore, is the only person allowed to look or modify the contents of the chest at the time it holds key.

The code between the lock and unlock calls the mutex, is referred to as a critical section. Mining the time spent in the critical section allows for great concurrency because it potentially reduces the amount of time other threads must wait to gain the lock.

[Mutex Type] There are 4 type but POSIX allows recursive and reader/writer style locks

Queueing: allows for fairness in lock acquisition by providing FIFO ordering to the arrival of lock requests. Such mutex may be slower due to increased overhead and the possibility of having to weak threads next in line that may be sleeping.

Reader/Writer: allows for multiple readers to acquire the lock simultaneously. If the existing readers have the lock, a writer request on the the lock will block until all readers have given up the lock. This can lead to writer Starvation [Stavation].

Recursive: allows a thread holding the lock to acquire the same lock again which may be necessary for recursive algorithm.

Potential Traps with Mutex

An important problem associated with mutexes is the possibility of deadlock. A program can deadlock if two (or more) threads have stoped execution or are spining permanently. For example, a simple deadlock situation: thread 1 locks A, thread 2 locks B, thread 1 wants to lock B and thread 2 wants lock A. Instant deadlock. You can prevent this from happening by making sure thread acquire locks in an agreed lock (i.e. preservation of lock ordering). Deadlock can also happen if threads do not unlock mutex properly.

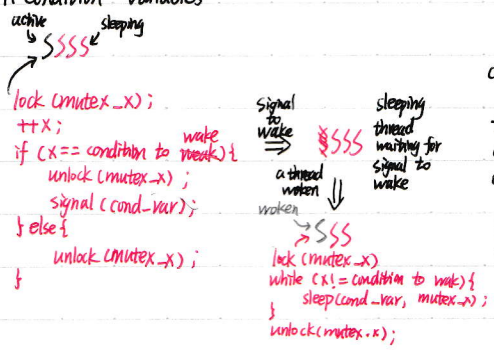
A race condition is when non-deterministic behavior results from threads accessing shared data or resources without following a defined synchronization protocol for serializing such access. This can result in erroneous outcomes that cause failure or inconsistent behaviour making race conditions particularly difficult to debug. In addition to incorrectly synchronized access to shared resources, library calls outside of your program's control are common culprits (罪魁祸首). Make sure you take steps within your program to enforce serial access to shared file descriptors or and other resource.

Another problem with mutexes is that contention for a mutex can lead to priority inversion. A high priority thread can wait behind a lower priority thread if the low priority thread if the lower priority thread holds a lock for which the higher priority thread is waiting. This can be eliminated /reduced by limiting the number of shared mutexes between different priority.

Atomic Operations

Atomic operations allow for concurrent algorithms and access to certain shared data types without the use of mutexes. For example, if there is sufficient compiler and system support, one can modify some variable (e.g., a 64-bit integer) within a multithread context without having to go through a locking protocol. Many atomic calls are non-portable and specific to the compiler and system.

Condition Variables



Condition variables allow threads to synchronize to a value of a shared resource. Typically, condition variables are used as a notification systems between threads.

For example, you could have a counter that once reaching a certain count, you would like for a thread to active. The thread that activates once the counter reaches the limit would wait on the condition variable. Active threads signal on this condition variable to notify other threads waiting/sleeping on the condition variable; thus causing a waiting thread to wakeup.

When waiting on condition variables, the wait should be inside a loop, not in a simple if statement because of spurious wakeups. You are not guaranteed that if a thread wakes up, it is the result of a signal or a broadcast call.

No.

Date

Barriers

Barriers are a method to synchronize a set of threads at some point in time by having all participating threads in the barrier wait until all threads have called the said barrier function. Thus, in essence, blocks all threads participating in the barrier until the slowest one reaches th

Semaphores

Semaphores are another type of synchronization primitive that come in two flavors: binary and counting. Binary semaphores act much like simple mutexes, while counting semaphores can behave as recursive mutexes. Counting seamphores can be initialized to any arbitrary value which should depe on how many resources you have available for that particular shared data. Many threads can obtain the lock simultaneously until the limit is reached. This is referred to as lock depth.

POSIX pthreads

[Creating pthreads] A pthread is represented by the type pthread_t. To create a thread

```
int pthread_create (pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void*), void *arg);
```

pthread_t * thread: the actual thread object that contains pthread id.
pthread_attr_t *attr: attributes to apply to this thread
void *(* start_routine)(void*): the function this thread executes.
Void *arg: arguments to pass to thread function above.

[Two important thread functions]

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

pthread_exit() terminates the thread and provides the pointer *value_ptr available to any pthread_join() call.
pthread_join() suspends the calling thread to wait for successful termination of the thread specified as the first argument pthread_t thre with an optional *value_ptr data passed from the terminating thread's call to pthread_exit().

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

#define NUM_THREAD 2
/* Create thread argument struct for thr_func() */
typedef struct _thread_data_t {
    int tid;
    double stuff;
} thread_data_t;

/* thread function */
void *thr_function (void *arg) {
    thread_data_t *data = (thread_data_t *) arg;
    printf("hello from thr_func, thread id: %d \n", data->tid);
    pthread_exit (NULL);
}

int main (int argc, char ** argv) {
    pthread_t thr[NUM_THREADS];
    int i, rc;
    /* create a thread_data_t argument array */
    thread_data_t thr_data[NUM_THREADS];

    /* create thread */
    for (i=0; i<NUM_THREADS; ++i) {
        thr_data[i].tid = i;
        if ((rc = pthread_create (&thr[i], NULL, thr_func, &thr_data[i]))) {
            fprint (stderr, "error: thread_create, rc: %d\n", rc);
            return EXIT_FAILURE;
        }
    }
    /* block until all threads complete */
    for (i=0; i<NUM_THREADS; ++i) {
        pthread_join (thr[i], NULL);
    }
    return EXIT_SUCCESS;
}
```

Function:

```
pthread_t thr[x]; ← 线程变量
pthread_create (&thr[x], NULL, thr_func, &data[x]);
```

线程变量 ↑ 方程名 ↑ 方程参数

```
#include <time.h>

void delay(int second)
{
    clock_t start = clock();
    while (clock()- start <= second*1000)
        ;
}
```

Thread function:

```
void *philosopher ( void * ID)
```

空指针

将 void* 型指针转变成 int* 型，并对 * 取指针指向的值。

方程内 variable 直接取指针值：

```
int n = *((int*)ID);
```

No.

Date

```
void vTaskResume ( TaskHandle_t xTaskToResume )
```
↑ Resume a suspended task

↓ Force a task to leave the Blocked state

```
BaseType_t xTaskAbortDelay ( TaskHandle_t xTask );
```

[Task Utilities]

```
TaskHandle_t xTaskGetCurrentTaskHandle (void);   // The handle of the currently running (calling) task
```
↑ 同type

```
xTaskGetHandle ( const char * pcNameToQuery );   // Looks up the handle of task from the task's name
```
* Take a relatively long time, should be stored for re-use

```
eTaskState eTaskGetState ( TaskHandle_t xTask );   // The state in which a task existed
```
↑ enum type { ① Ready : eReady ② Blocked ⑤ Deleted ② Running : eRunning ④ Suspended

```
char * pcTaskGetName ( TaskHandle_t xTaskToQuery );   // Look up the name of a task from the task's hand
volatile TickType_t xTaskGetTickCount (void); / xTaskGetTickCountFromISR (void);
```

queue.h

```
QueueHandle_t xQueueCreate ( UBaseType_t uxQueueLength,   // The maximum number of items
```
↑ Create a new queue and returns a handle

```
                             UBaseType_t uxItemSize )   // The size, in bytes for each item
void vQueueDelete ( QueueHandle_t xQueue );
```

```
BaseType_t xQueueSend ( QueueHandle_t xQueue,   // The handle to the queue on which the item is to be posted
```
↑ Post an item on a queue return pdTRUE

```
                        const void * pvItemToQueue,   // A pointer to the item that is to be placed
                        TickType_t xTicksToWait );   // The maximum amount of time the task should block waiting for space to become available on the queue
```

```
BaseType_t xQueueReceive ( QueueHandle_t xQueue,
```
↑ Receive an item from a queue

```
                           void *pvBuffer,   // Pointer to the buffer into which the received item will be copied
                           TickType_t xTicksToWait );
BaseType_t xQueueReset ( QueueHandle_t xQueue );
BaseType_t xQueuePeek ( QueueHandle_t xQueue,
```
↑ Receive an item from a queue without removing the item

```
                        void *pvBuffer,
                        TickType_t xTicksToWait );
```

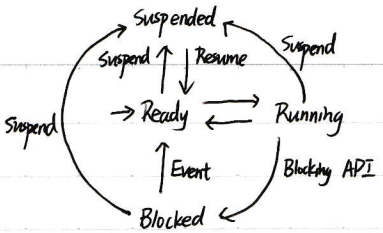
vTaskDelay (mSeconds2Ticks(100));

No.

Date

FreeRTOS

Application Programming Interface



task.h

TaskHandle_t Type by which tasks are referenced

BaseType_t xTaskCreate (TaskFunction_t pvTaskCode, // Pointer to the task entry function
const char * const pcName, // A descriptive name for the task
unsigned short usStackDepth, // The number of words to allocate for use
void * pvParameters, // Task's parameter
UBaseType_t uxPriority, // The priority at which the created task will execute (configMAX_PRIORITIES)
TaskHandle_t * pxCreatedTask); // Used to pass a handle to the created task (NULL) (xHandle 넣는곳!!!)

↑ Create a new task and add it to the list of tasks that are ready to run

void vTaskDelete (TaskHandle_t xTask); // Passing NULL will cose the calling task to be deleted.

↑ Remove a task from the RTOS kernels management

Task Creation Control

void vTaskDelay (const TickType_t xTicksToDelay);

↖ specifics a time at which the task wishes to unblock. ※Not EXACT Time. Other task and interrupt will effect!

eg.
```
void vTaskFunction ( void * pvParameters )
{
    const TickType_t xDelay = 500 / portTICK_PERIOD_MS;   // Used to calculate the real time
    for( ;; )
    {
        vToggleLED();
        vTaskDelay ( xDelay );
    }
}
```

~~void vTaskDelayUntil~~ ↙ Delay a task until a SPECIFIC time

void vTaskDelayUntil (TickType_t *pxPreviousWakeTime, // MUST be initialised with the current time (Then AUTO)
const TickType_t xTimeIncrement); // The cycle time period

Get the current time ↙ xLastWakeTime = xTaskGetTickCount();

↙ Obtain the priority of a task

UBaseType_t uxTaskPriorityGet (TaskHandle_t xTask);

↙ Set the priority of any task

void vTaskPrioritySet (TaskHandle_t xTask, // The handle which is store when creating the task
UBaseType_t uxNewPriority);

↙ Suspend any task

void vTaskSuspend (TaskHandle_t xTaskToSuspend);

Linux C
uint32_t 32位无符号类型数
uint64_t

No.
Date

FRDM-KL46Z

LED Configuration

bit12 → SCGC5[12] PortD Clock

```
#define SIM_SCGC5_PORTD_MASK 0x1000u = 1,0000,0000,0000

void InitLED(void){
    SIM->SCGC5 |= SIM_SCGC5_PORTD_MASK;   // |= 写 a|b
    PORTD->PCR[5] = PORT_PCR_MUX(1u);
    PTD->PDDR |= (1u<<5);                 // 1u<<5 , 1u表示 unsigned decimal 1 , <<5表示左移5位
}
```

(((uint32_t)(x)) << PORT_PCR_MUX_Shift) & PORT_PCR_MUX_MASK)
(8) (0x700u)
⇒ when x=1u , PORT_PCR_MUX(1u) = ...001 00000000 (bit[8]=1)

① System Integration Module (SIM) /p216
Gives control of which peripherals are clocked (leaving them without access to the clock we save power, but leave them inoperable)
- System clock gating control register 5 (SIM_SCGC5)
  This register holds the clock enable for the various ports on the MCU
  ※ If use any pins on a port, must be enabled

SIM_SCGC5 (总共32 bits) :
- SIM_SCGC5[9] : Port A Clock Gate Control
- SIM_SCGC5[10] : Port B Clock Gate Control
- SIM_SCGC5[11] : Port C Clock Gate Control
- SIM_SCGC5[12] : Port D Clock Gate Control
- SIM_SCGC5[13] : Port E Clock Gate Control

{ 0: Clock disabled
{ 1: Clock enabled

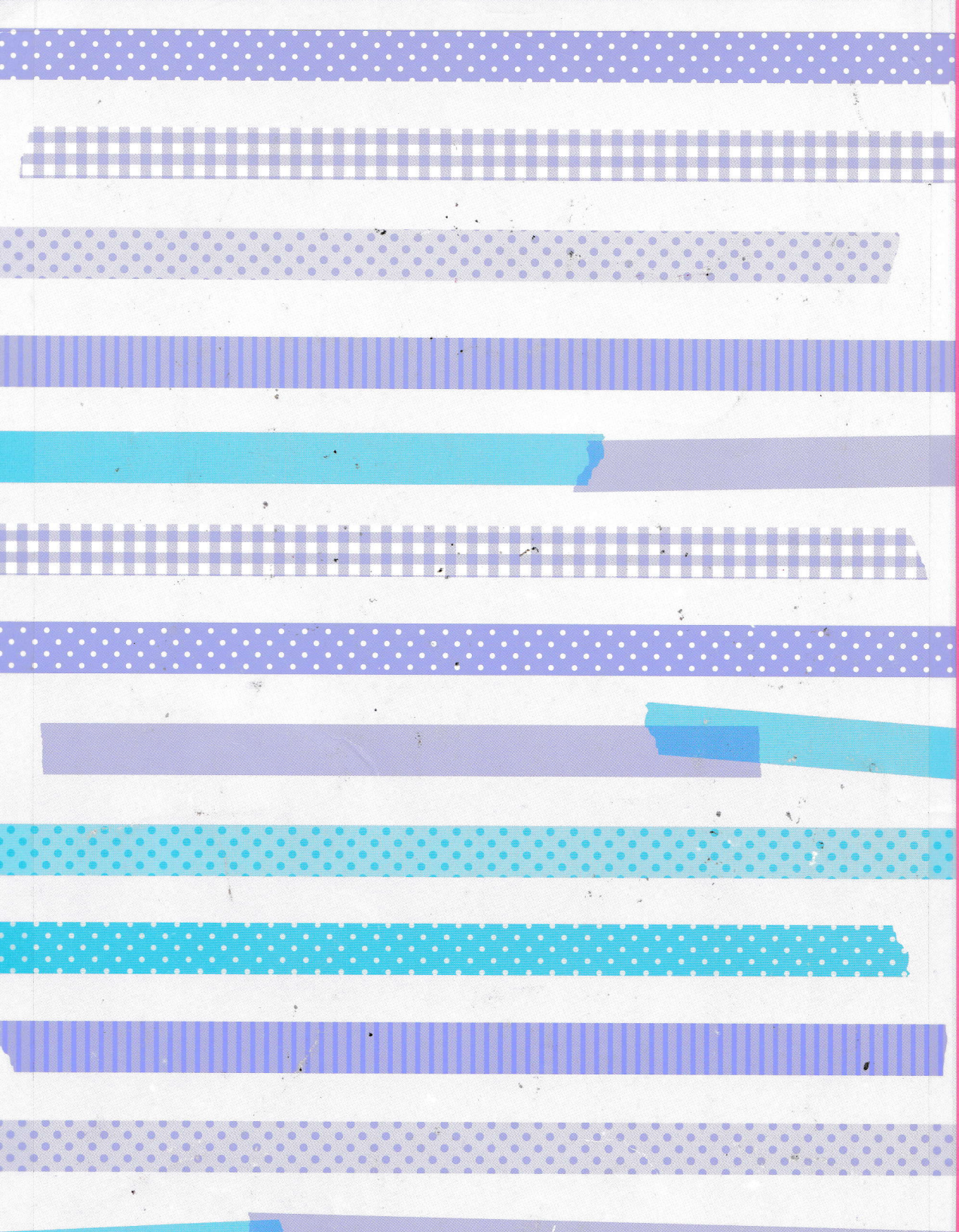
② Port control and interrupts (PORT) /194
This pin register determines which pins are used for which peripheral
- Pin control Register n (PORTx_PCRn) x=A,B,C,D ; n=0,1,...,31 PORTD → PCR[5]
  PORTn_PCRx[10-8] : Pin Mux Control
  { 000: Pin disabled
  { 001: Alternative 1 (GPIO)
  0b111 : Alternative 2~7 (chip-specific)

③ General Purpose Input/Output (GPIO) GPIO contains : PTA, PTB, PTC, PTD, PTE (length : 32 bits)
Once a pin is configured as GPIO, this register determines whether it is an input or an output / read from and write to
- Port data direct register (GPIO_PDDR)
  This register configures GPIO as inputs or outputs ※ { 0: input { 1: output
  PTD → PDDR |= (1u<<5);
- Port data output register (GPIO_PDOR)
  This register will change the output of a pin to the value given PTD → PDOR &= ~((uint32_t)(1u<<5));
- Port data set register (GPIO_~~PDSR~~PSOR)
  Input 0 will not change the current value of the pin, while inputing 1 will set the bit to 1
- Port data clear register (GPIO_PCOR)
  Input 0 will not change the current value of the pin, while input of 1 will ~~set~~ clear the bit to 0
- Port data toggle register (GPIO_PTOR)
  ..., while input of 1 will inverse the current state
- Port data input register (GPIO_PDIR)
  This register returns the value of an input port in a 32 bit format.
  ↳ PCR[Pin] 的 10-8位设置001后, 还需设置输入的 PS位 (Pull Select) 和 PE位 (Pull Enable) { bit 0 PS { 0 Pull down { 1 Pull up ; { bit 1 PE { 0 Disable { 1 Enable

Accelerator

I2C Protocol

		(I2C0)
SDA	(Data)	PTE25
SCL	(Clock)	PTE24



Campus 无线装订本
B5 40页
WCN-CNB1430
6 937748 309932
MADE IN CHINA

国誉商业(上海)有限公司
http://www.kokuyo.cn/st/
上海市奉贤区人杰路128号
TEL：400-820-0798　FAX：021-3255-8508
产地：上海市　QB/T 1438-2007合格
B5　252×179mm

● 采用日本进口纸张，书写顺滑，牢固不掉页。