Concurrent Computing (Operating Systems)

Daniel Page

Department of Computer Science,
University Of Bristol,
Merchant Venturers Building,
Woodland Road,
Bristol, BS8 1UB. UK.
⟨Daniel.Page⟩

February 9, 2018

Keep in mind there are *two* PDFs available (of which this is the latter):

1. a PDF of examinable material used as lecture slides, and
2. a PDF of non-examinable, extra material:
   - the associated notes page may be pre-populated with extra, written explaination of material covered in lecture(s), plus
   - anything with a "grey'ed out" header/footer represents extra material which is useful and/or interesting but out of scope (and hence not covered).

Notes:

Notes:

- Question:
  1. which *algorithm* do you select: $O(n)$ or $O(\log n)$
  2. which *implementation* do you select: $O(n)$ or $O(\log n)$

Notes:

- Rob Pike [1] formulated 5 rules of programming: see

  http://users.ece.utexas.edu/~adnan/pike.html

  for example. In essence this slide captures rule #3, but all of them are relevant to the point being made.

COMS20001 additional content

- Question:
  1. which *algorithm* do you select: $O(n)$ or $O(\log n)$
  2. which *implementation* do you select: $O(c_1 \cdot n)$ or $O(c_2 \cdot \log n)$

Notes:

- Rob Pike [1] formulated 5 rules of programming: see

  http://users.ece.utexas.edu/~adnan/pike.html

  for example. In essence this slide captures rule #3, but all of them are relevant to the point being made.

- Question:
  1. which *algorithm* do you select: $O(n)$ or $O(\log n)$
  2. which *implementation* do you select: $O(20 \cdot 10)$ or $O(200 \cdot \log 10)$

Notes:

- Rob Pike [1] formulated 5 rules of programming: see

  http://users.ece.utexas.edu/~adnan/pike.html

  for example. In essence this slide captures rule #3, but all of them are relevant to the point being made.

COMS20001 additional content

- Question:
  1. which *algorithm* do you select: $O(n)$ or $O(\log n)$
  2. which *implementation* do you select: $O(20 \cdot 10)$ or $O(200 \cdot \log 10)$

- Conclusion:
  - effective use of data structures and algorithms theory will **always** offer a starting point,
  - *but*, keep in mind that

$$f(n) = O(g(n)) \quad \Rightarrow \quad \exists\, c > 0 \text{ st. for } n > n',\ |f(n)| \leq c \cdot |g(n)|.$$

  st. in practice

  1. *if*, for example, n is small and/or c is large, the theoretical analysis may not be robust, *plus*
  2. we might tolerate or prefer less optimal output *if* it can be produced more efficiently,

  i.e., the "best" data structure or algorithm *may* not be the most *appropriate* choice.

Notes:

- Rob Pike [1] formulated 5 rules of programming: see

  http://users.ece.utexas.edu/~adnan/pike.html

  for example. In essence this slide captures rule #3, but all of them are relevant to the point being made.

- Question:
  1. which *algorithm* do you select: $O(n)$ or $O(\log n)$
  2. which *implementation* do you select: $O(20 \cdot 10)$ or $O(200 \cdot \log 10)$
- Conclusion:
  - effective use of data structures and algorithms theory will **always** offer a starting point,
  - *but*, keep in mind that

$$f(n) = O(g(n)) \quad \Rightarrow \quad \exists\, c > 0 \text{ st. for } n > n',\ |f(n)| \leq c \cdot |g(n)|.$$

  st. in practice

  1. *if*, for example, n is small and/or c is large, the theoretical analysis may not be robust, *plus*
  2. we might tolerate or prefer less optimal output *if* it can be produced more efficiently,

  i.e., the "best" data structure or algorithm *may* not be the most *appropriate* choice.
- Conclusion: the devil is *very much* in the detail.

Notes:

- Rob Pike [1] formulated 5 rules of programming: see

  `http://users.ece.utexas.edu/~adnan/pike.html`

  for example. In essence this slide captures rule #3, but all of them are relevant to the point being made.

- Question: what might contribute to theoretical analysis (i.e., the big-O)?

Notes:

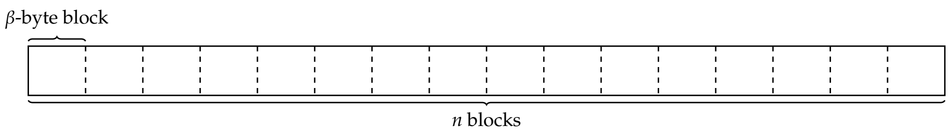
COMS20001 additional content

- Question: what might contribute to theoretical analysis (i.e., the big-O)?
- Answer:
  1. the expression form:
     - algorithm,
     - data structure,
     - ...
  2. the problem size:
     - user behaviour,
     - system behaviour,
     - ...
  3. the constant factors:
     - quality of programming,
     - quality of compilation,
     - quality of run-time support,
     - overhead of abstraction layers between program and platform,
     - latency of instruction execution,
     - latency of access to memory hierarchy,
     - ...

Notes:

Examples (1)
Free space management

- Problem: the kernel needs a mechanism for **free space management**, i.e., to manage allocation of n blocks (each of β bytes).



e.g.,

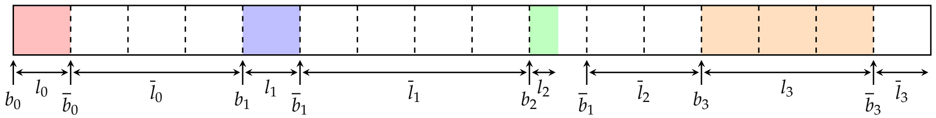
$$\begin{array}{lcrcr} \text{heap segment} & \rightsquigarrow & \beta & \simeq & 1\text{B} \\ \text{disk blocks} & \rightsquigarrow & \beta & \simeq & 512\text{B} \\ \text{virtual memory pages} & \rightsquigarrow & \beta & \simeq & 4096\text{B} \end{array}$$

Notes:

- The selection of an allocation policy depends on a variety of factors, including the anticipated form of allocation requests; this demands an understanding of the behaviour of processes. Beyond this, it can make sense to rely on simulation [5] to judge when/where each policy might be useful.
- There are numerous more advanced allocation policies not covered here. Two of the more popular are **slab allocation** [6] and **buddy allocation** [7]. The idea of the former is essentially to pre-allocate pools (that can be thought of as *multiple* data structures, one associated with each each pool) of specific size to efficiently satisfy requests of a common or expected size, and then defer to a more general allocation policy for less common or generic requests. The idea of the latter is to optimise merge operations, sub-dividing the space using a tree of regions each whose size is a power-of-two; when a region at a given level is deallocated, it is efficient to check if the "buddy" region (at the same level of the tree) is unused, and therefore merge them.

Examples (1)
Free space management

- Problem: the kernel needs a mechanism for **free space management**, i.e., to manage allocation of n blocks (each of β bytes).



or, put another way, we need to

1. maintain a data structure of **allocation records**, e.g.,

$$\alpha = \langle \alpha_0 = (b_0, l_0), \alpha_1 = (b_1, l_1), \ldots, \alpha_{m-1} = (b_{m-1}, l_{m-1}) \rangle,$$

that can be manipulated via supporting algorithms such as

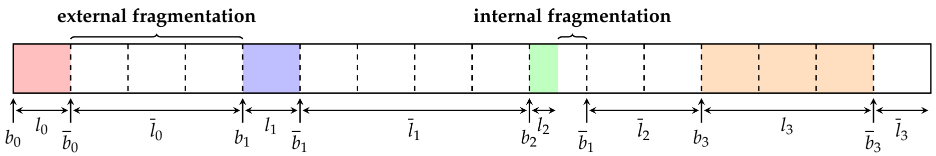
- ALLOCATE : add a new allocation record
- DEALLOCATE : remove an existing allocation record
- SPLIT : split one existing allocation record into several records
- MERGE : merge several existing allocation records into one record
- DEFRAGMENT : reorganise existing allocation records to reduce fragmentation

Notes:

- The selection of an allocation policy depends on a variety of factors, including the anticipated form of allocation requests; this demands an understanding of the behaviour of processes. Beyond this, it can make sense to rely on simulation [5] to judge when/where each policy might be useful.
- There are numerous more advanced allocation policies not covered here. Two of the more popular are **slab allocation** [6] and **buddy allocation** [7]. The idea of the former is essentially to pre-allocate pools (that can be thought of as *multiple* data structures, one associated with each each pool) of specific size to efficiently satisfy requests of a common or expected size, and then defer to a more general allocation policy for less common or generic requests. The idea of the latter is to optimise merge operations, sub-dividing the space using a tree of regions each whose size is a power-of-two; when a region at a given level is deallocated, it is efficient to check if the "buddy" region (at the same level of the tree) is unused, and therefore merge them.

Examples (1)
Free space management

- Problem: the kernel needs a mechanism for **free space management**, i.e., to manage allocation of n blocks (each of β bytes).



or, put another way, we need to

2. satisfy **allocation requests**, e.g.,

$$\rho = \langle \rho_0, \rho_1, \ldots \rangle$$

where say some $\rho_i = 2 \cdot \beta$, using an **allocation policy**

- **first fit** : search for first j st. $\bar{l}_j \geq \rho_i$ $\Rightarrow$ allocate at $\bar{b}_j$
- **best fit** : search for a j st. $\bar{l}_j \geq \rho_i$ and $|\bar{l}_j - \rho_i|$ is minimised $\Rightarrow$ allocate at $\bar{b}_j$
- **worst fit** : search for a j st. $\bar{l}_j \geq \rho_i$ and $|\bar{l}_j - \rho_i|$ is maximised $\Rightarrow$ allocate at $\bar{b}_j$

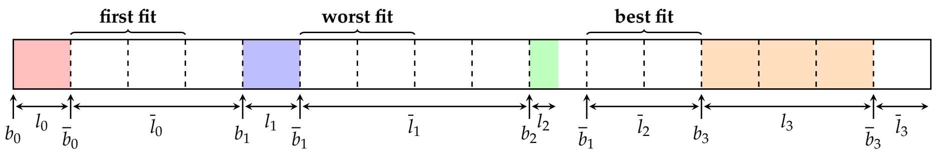
whose goal is to maximising utilisation (noting one cannot *know* a ρ_j for $j > i$ at time i).

Notes:

- The selection of an allocation policy depends on a variety of factors, including the anticipated form of allocation requests; this demands an understanding of the behaviour of processes. Beyond this, it can make sense to rely on simulation [5] to judge when/where each policy might be useful.
- There are numerous more advanced allocation policies not covered here. Two of the more popular are **slab allocation** [6] and **buddy allocation** [7]. The idea of the former is essentially to pre-allocate pools (that can be thought of as *multiple* data structures, one associated with each each pool) of specific size to efficiently satisfy requests of a common or expected size, and then defer to a more general allocation policy for less common or generic requests. The idea of the latter is to optimise merge operations, sub-dividing the space using a tree of regions each whose size is a power-of-two; when a region at a given level is deallocated, it is efficient to check if the "buddy" region (at the same level of the tree) is unused, and therefore merge them.

Examples (1)
Free space management

- Problem: the kernel needs a mechanism for **free space management**, i.e., to manage allocation of n blocks (each of β bytes).



or, put another way, we need to

2. satisfy **allocation requests**, e.g.,

$$\rho = \langle \rho_0, \rho_1, \ldots \rangle$$

where say some $\rho_i = 2 \cdot \beta$, using an **allocation policy**

- **first fit** : search for first j st. $\bar{l}_j \geq \rho_i$ $\Rightarrow$ allocate at $\bar{b}_j$
- **best fit** : search for a j st. $\bar{l}_j \geq \rho_i$ and $|\bar{l}_j - \rho_i|$ is minimised $\Rightarrow$ allocate at $\bar{b}_j$
- **worst fit** : search for a j st. $\bar{l}_j \geq \rho_i$ and $|\bar{l}_j - \rho_i|$ is maximised $\Rightarrow$ allocate at $\bar{b}_j$

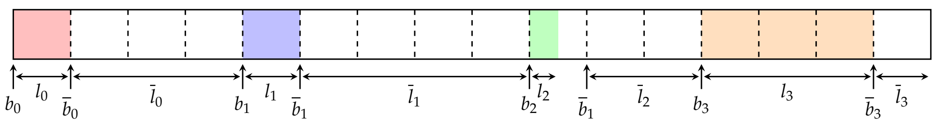
whose goal is to maximising utilisation (noting one cannot *know* a ρ_j for $j > i$ at time i).

Notes:

- The selection of an allocation policy depends on a variety of factors, including the anticipated form of allocation requests; this demands an understanding of the behaviour of processes. Beyond this, it can make sense to rely on simulation [5] to judge when/where each policy might be useful.
- There are numerous more advanced allocation policies not covered here. Two of the more popular are **slab allocation** [6] and **buddy allocation** [7]. The idea of the former is essentially to pre-allocate pools (that can be thought of as *multiple* data structures, one associated with each each pool) of specific size to efficiently satisfy requests of a common or expected size, and then defer to a more general allocation policy for less common or generic requests. The idea of the latter is to optimise merge operations, sub-dividing the space using a tree of regions each whose size is a power-of-two; when a region at a given level is deallocated, it is efficient to check if the "buddy" region (at the same level of the tree) is unused, and therefore merge them.

Examples (1)
Free space management

- Problem: the kernel needs a mechanism for **free space management**, i.e., to manage allocation of n blocks (each of β bytes).



© Daniel Page ( )
Concurrent Computing (Operating Systems) git # 1d9e4fa @ 2018-02-08
University of BRISTOL

Notes:

- The selection of an allocation policy depends on a variety of factors, including the anticipated form of allocation requests; this demands an understanding of the behaviour of processes. Beyond this, it can make sense to rely on simulation [5] to judge when/where each policy might be useful.
- There are numerous more advanced allocation policies not covered here. Two of the more popular are **slab allocation** [6] and **buddy allocation** [7]. The idea of the former is essentially to pre-allocate pools (that can be thought of as *multiple* data structures, one associated with each each pool) of specific size to efficiently satisfy requests of a common or expected size, and then defer to a more general allocation policy for less common or generic requests. The idea of the latter is to optimise merge operations, sub-dividing the space using a tree of regions each whose size is a power-of-two; when a region at a given level is deallocated, it is efficient to check if the "buddy" region (at the same level of the tree) is unused, and therefore merge them.

Examples (2)
Free space management

- How?!
  - we have numerous options, e.g.,

	Average case				Worst case			
	Access	Search	Insert	Delete	Access	Search	Insert	Delete
Array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Queue	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Singly-linked list	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Doubly-linked list	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Skip list	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Hash table		$\Theta(1)$	$\Theta(1)$	$\Theta(1)$		$O(n)$	$O(n)$	$O(n)$
Binary tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Red-Black tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$

  - *but* careful analysis is required wrt. questions such as
    - how to these operations relate to the support algorithms and/or allocation policy, e.g., which dominate?
    - what is the probability of encountering, e.g., worst case behaviour?
    - what is the impact of encountering, e.g., worst case behaviour?
    - can we tolerate, and what is the impact of probabilistic vs. precise behaviour?
    - how large is n, and what bounds can we expect on m?
    - what are the constant factors for these options?
    - ...

http://www.bigocheatsheet.com

Notes:

Examples (3)
Scheduling queues

Quote

Computer manufacturers of the 1960s estimated that more than 25 percent of the running time of their computers was spent on sorting, when all their customers were taken into account. In fact, there were many installations in which the task of sorting was responsible for more than half of the computing time. From these statistics we may conclude that either

- *there are many important applications of sorting, or*
- *many people sort when they shouldn't, or*
- *inefficient sorting algorithms have been in common use.*

– Knuth [3, Page 3]

Notes:

Examples (4)
Scheduling queues

- Problem: the kernel needs to
  1. maintain a **process table** (of PCBs),
  2. maintain various **scheduling queues** which track processes, e.g.,
     - $1 \times$ **ready queue** : processes that can be executed
     - $n \times$ **waiting queue** : processes whose execution is blocked

     and
  3. sort said queues, e.g., to realise a priority-based scheduling algorithm.

Notes:

Examples (5)
Scheduling queues

- How?!
  - we have numerous options, e.g.,

	Best case	Average case	Worst case
Quicksort	$\Omega(n\log(n))$	$\Omega(n\log(n))$	$O(n^2)$
Mergesort	$\Omega(n\log(n))$	$\Omega(n\log(n))$	$O(n\log(n))$
Heapsort	$\Omega(n\log(n))$	$\Omega(n\log(n))$	$O(n\log(n))$
Bubble sort	$\Omega(n)$	$\Omega(n^2)$	$O(n^2)$
Insertion sort	$\Omega(n)$	$\Omega(n^2)$	$O(n^2)$
Selection sort	$\Omega(n^2)$	$\Omega(n^2)$	$O(n^2)$

  - *but* careful analysis is required wrt. questions such as
    - what is the probability of encountering, e.g., worst case behaviour?
    - what is the impact of encountering, e.g., worst case behaviour?
    - can we tolerate, and what is the impact of probabilistic vs. precise behaviour?
    - how large is n?
    - what are the constant factors for these options?
    - ...

http://www.bigocheatsheet.com

Notes:

Linux data structures and algorithms: (linked-)lists (1)

- Example: the Linux list implementation

Listing (`linux-2.6.10/include/linux/list.h`)

```c
struct list_head {
        struct list_head *next, *prev;
};
```

is

1. doubly-linked,
2. cyclically-linked, and
3. *invasive*
   - – non-standard vs. usual, non-invasive alternative
   - \+ don't need "list for X" *and* "list for Y"
   - \+ don't need opaque type for payload
   - \+ allows (fairly) localised access (e.g., within a table)

Notes:

Linux data structures and algorithms: (linked-)lists (1)

- Example: the Linux list implementation

Listing (`linux-2.6.10/include/linux/list.h`)

```c
struct list_head {
        struct list_head *next, *prev;
};
```

exhibits some attractive design features wrt.

1. generalisation:

   `list_add` ⇒ supports list or stack
   `list_add_tail` ⇒ supports queue

2. modularity:

   `__list_add` ⇒ basic operation used by `list_add` *and* `list_add_tail`

3. abstraction:

   `list_for_each` ⇒ iterate through list

4. robustness (via defensive programming):

   `list_del_init` ⇒ delete node *and* reinitialise (i.e., sanitise) pointers

Notes:

Linux data structures and algorithms: (linked-)lists (2)

- Example: given

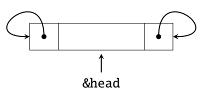
Listing (C)

```c
struct list_head head;

struct object_t {
  struct list_head list;
  ...
};

struct object_t object0;
struct object_t object1;
```

then

```c
INIT_LIST_HEAD( &head );
```

yields



Notes:

Linux data structures and algorithms: (linked-)lists (2)

- Example: given

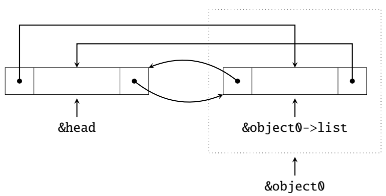
Listing (C)

```c
struct list_head head;

struct object_t {
  struct list_head list;
  ...
};

struct object_t object0;
struct object_t object1;
```

then

```c
list_add( &object0->list, &head );
```

yields



Notes:

Linux data structures and algorithms: (linked-)lists (2)

- **Example**: given

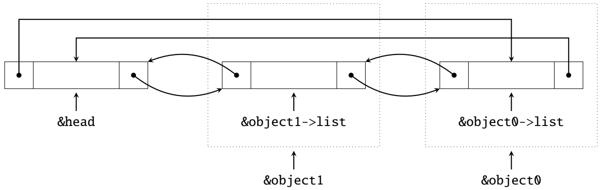
Listing (C)

```c
struct list_head head;

struct object_t {
  struct list_head list;
  ...
};

struct object_t object0;
struct object_t object1;
```

then

```c
list_add( &object1->list, &head );
```

yields



Notes:

Linux data structures and algorithms: (linked-)lists (2)

- **Example**: given

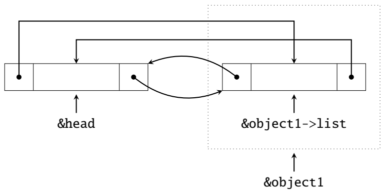
Listing (C)

```c
struct list_head head;

struct object_t {
  struct list_head list;
  ...
};

struct object_t object0;
struct object_t object1;
```

then

```c
list_del( &object0->list );
```

yields



Notes:

Linux data structures and algorithms: (linked-)lists (2)

- Example: given

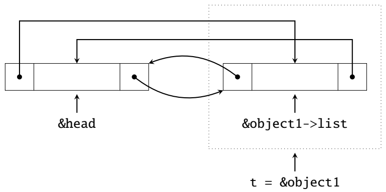
Listing (C)

```c
struct list_head head;

struct object_t {
  struct list_head list;
  ...
};

struct object_t object0;
struct object_t object1;
```

then

```c
object_t* t = list_entry( &head->next );
```

yields



Notes:

Linux data structures and algorithms: bit-map (1)
Concept

Data Structure

A **bit-map** is a data structure used to capture membership of (vs. content, i.e., the objects in) a collection:

- we assume a universe $\mathcal{U}$ of objects that can be identified via an integer index,
- a bit-map X is a sequence of n bits where if

$$X_i = \begin{cases} 0 & \text{then the } i\text{-th object is not a member} \\ 1 & \text{then the } i\text{-th object is} \quad\ \text{a member} \end{cases}$$

  for $0 \le i < n$, but
- we specialise this to suit the processor; defining $n = n' \cdot w$ means the bit-map is a sequence of n' words, each of w bits.

Example

If we set $n' = 2$ and $w = 8$ then $n = 2 \cdot 8 = 16$, and

$$\begin{array}{rcl} X & = & \langle \underbrace{0,0,0,1,0,1,0,0}_{\text{0-th word}}, \underbrace{0,0,1,0,1,0,0,0}_{\text{1-th word}} \rangle \\ & \equiv & \langle 00101000_{(2)}, 00010100_{(2)} \rangle \\ & \equiv & \texttt{\{ 0x28, 0x14 \}} \end{array}$$

st.

$$\begin{array}{lcl} X_0 = 0 & \Rightarrow & \text{the 0-th object is not a member} \\ X_3 = 1 & \Rightarrow & \text{the 3-rd object is} \quad\ \text{a member} \end{array}$$

Notes:

- It is reasonable to use the terms **bit-vector** or **bit-set** as synonyms for bit-map (as an aside, the unhyphenated **bitmap** acts as a fair alternative as well); in the former case the vector aspect suggests this is a sequence, in the latter case the set aspect hints that we can support efficient set operations (and, by definition, we know that since $X_i \in \{0,1\}$ there cannot be multiple instances of the object in the collection).
- To allow efficient implementation of a bit-map, it is common to fix n and assume it is constant; the alternative would be to permit an *extensible* bit-map, implying (more) involved memory management and disallowing some compile-time optimisations (since the compiler will have to assume n is a variable, rather than a constant).

- Why?!
  1. basic operations

test	i-th object	$\Rightarrow$	extract	i-th bit of X
add	i-th object	$\Rightarrow$	set	i-th bit of X
remove	i-th object	$\Rightarrow$	clear	i-th bit of X

  are *very* efficient,

Notes:

Linux data structures and algorithms: bit-map (2)
Basic operations

- Why?!

  2. we can support more advanced, set-like operations, e.g.,

complement	of X	$\Rightarrow$	$\overline{X}$	$\Rightarrow$	NOT	each i-th bit of X
intersection	of X and Y	$\Rightarrow$	$X \cap Y$	$\Rightarrow$	AND	each i-th bit of X and Y
union	of X and Y	$\Rightarrow$	$X \cup Y$	$\Rightarrow$	OR	each i-th bit of X and Y

  naturally and efficiently.

© Daniel Page (csdsp@bristol.ac.uk)
Concurrent Computing (Operating Systems) git # 1d9e4fa @ 2018-02-08
University of BRISTOL

Notes:

Linux data structures and algorithms: bit-map (3)

Basic operations

Listing (linux-2.6.10/include/linux/types.h)

```
#define BITS_TO_LONGS(bits) \
        (((bits)+BITS_PER_LONG-1)/BITS_PER_LONG)
#define DECLARE_BITMAP(name,bits) \
        unsigned long name[BITS_TO_LONGS(bits)]
```

Listing (linux-2.6.10/include/linux/arch/asm-arm/bitops.h)

```
static inline int __test_bit(int nr, const volatile unsigned long * p)
{
        return (p[nr >> 5] >> (nr & 31)) & 1UL;
}
```

Listing (linux-2.6.10/include/linux/arch/asm-arm/bitops.h)

```
static inline void __set_bit(int nr, volatile unsigned long *p)
{
        p[nr >> 5] |= (1UL << (nr & 31));
}

static inline void __clear_bit(int nr, volatile unsigned long *p)
{
        p[nr >> 5] &= ~(1UL << (nr & 31));
}
```


Notes:

Linux data structures and algorithms: bit-map (4)

Advanced operations

- It's useful to support
  1. one of more of

find first set	$\Rightarrow$	FFS
find last set	$\Rightarrow$	FLS
find first zero	$\Rightarrow$	FFZ
find last zero	$\Rightarrow$	FLZ
count leading ones	$\Rightarrow$	CLO
count trailing ones	$\Rightarrow$	CTO
count leading zeros	$\Rightarrow$	CLZ
count trailing zeros	$\Rightarrow$	CTZ

     which allows search for objects in (or not in) X, and
  2. a **population count**, i.e.,

$$\textsc{pop}(x) = \mathcal{H}(x),$$

     which allows computation of the cardinality of (or number of objects in) X

  and, again, this can be done in an efficient manner ...

Notes:

- As well as direct hardware support, these operations crop up as GCC built-in functions (or intrinsics, e.g., `__builtin_ffs`), and within POSIX (e.g., [4, Page 847] is a definition for `ffs`, as included in `strings.h`).

Linux data structures and algorithms: bit-map (5)
Advanced operations

- Some subtle details suggest a design space of options, e.g., wrt.
  1. terminology:

$$
X = 52_{(10)} \mapsto 00110100_{(2)} \simeq
\begin{array}{cccccccc}
\multicolumn{2}{c}{\xrightarrow{\text{last}}} & & & & & \multicolumn{2}{c}{\xleftarrow{\text{first}}} \\
x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 \\
\hline
\multicolumn{1}{|c|}{0} & \multicolumn{1}{c|}{0} & \multicolumn{1}{c|}{1} & \multicolumn{1}{c|}{1} & \multicolumn{1}{c|}{0} & \multicolumn{1}{c|}{1} & \multicolumn{1}{c|}{0} & \multicolumn{1}{c|}{0} \\
\hline
\text{bit } 7 = w-1 & \text{bit } 6 & \text{bit } 5 & \text{bit } 4 & \text{bit } 3 & \text{bit } 2 & \text{bit } 1 & \text{bit } 0 \\
\multicolumn{2}{c}{\xrightarrow{\text{leading}}} & & & & & \multicolumn{2}{c}{\xleftarrow{\text{trailing}}}
\end{array}
$$

  noting that a two's-complement representation of integers is typically assumed,
  2. interface:
     1. practicality $\Rightarrow$ operations for w-bit x rather than n-bit X
     2. orthogonality $\Rightarrow$ don't need *every* operation, e.g., $\textsc{ffz}(x) = \textsc{ffs}(\neg x)$

  and
  3. semantics:
     1. indexing $\Rightarrow$ are bits in x referred to via 0-indexing or 1-indexing?
     2. special-cases $\Rightarrow$ what happens if x is 0?

  where different implementations *do* make different (and so incompatible) choices.

Notes:

- Clearly we have that

$$
\begin{array}{lclcl}
\textsc{clo}(-1) & = & \textsc{clo}(1\ldots11) & = & w \\
\textsc{cto}(-1) & = & \textsc{cto}(1\ldots11) & = & w \\
\textsc{clz}(0) & = & \textsc{clz}(0\ldots00) & = & w \\
\textsc{ctz}(0) & = & \textsc{clz}(0\ldots00) & = & w \\
\textsc{ffs}(x) & = & \textsc{ffz}(\neg x) & & \\
\textsc{fls}(x) & = & \textsc{flz}(\neg x) & & \\
\textsc{clo}(x) & = & \textsc{clz}(\neg x) & & \\
\textsc{cto}(x) & = & \textsc{ctz}(\neg x) & &
\end{array}
$$

  but beyond this, *lots* of relationships exist between the operations; this implies we could provide just a few of them, yet support them all by synthesising those we don't provide.
  For example, *if* we use 1-indexing for bits, then $\textsc{ffs}(x) = w - \textsc{clz}(x \wedge (-x))$ so we can provide CLZ and synthesise FFS. Why is this relationship true? Assuming $x \neq 0$, the expression $r = x \wedge (-x)$ means $r_i = 0$, except r_j for the smallest j st. $x_j = 1$ (i.e., except the least-significant 1 bit in x) which is set to 1. Consider this example

$$
\begin{array}{lcrcl}
x & = & 52_{(10)} & \mapsto & 00110100_{(2)} \\
-x & = & -52_{(10)} & \mapsto & 11001100_{(2)} \\
x \wedge (-x) & = & & \mapsto & 00000100_{(2)}
\end{array}
$$

  where $w = 8$ and the least-significant 1 bit in x is where $j = 2$. Since $-x = \neg x + 1$, using two's-complement, each least-significant 0 in x becomes a 1. The term $\neg x$ produces a run (i.e., a sub-sequence of consecutive) least-significant 1 bits; by then adding 1, the resulting carry will a) turn each bin in this run to 0, and b) turn the first 0 bit (i.e., what was the least-significant 1 bit in x) to 1. So, if we apply CLZ to

$$x \wedge (-x) \mapsto 00000100_{(2)}$$

  we get 5 because the 5 MSBs are 0; therefore $w - 5 = 8 - 5 = 3$ gives the same as $\textsc{ffs}(x)$ (given we use 1-indexing).

Linux data structures and algorithms: bit-map (6)
Advanced operations: FFS

- Solution #1: utilise direct hardware support, e.g.,
  1. ARM $\Rightarrow$ `clz` (i.e., CLZ)
  2. x86 $\Rightarrow$ `bsf` (i.e., CLZ)

Notes:

- Solution #2: perform direct, bit-by-bit process.

Algorithm

```
algorithm FFS(x) begin
  if x = 0 then
    return 0
  else
    i ← 1
    while x ∧ 1 = 0 do
      i ← i + 1, x ← x ≫ 1
    end
    return i
  end
end
```

Notes:

- Solution #3: employ a pre-computation style approach.

Algorithm

1. first pre-compute a table st.
   $$T[x] = \textsc{ffs}(x)$$
   for $0 \leq x < 2^k - 1$ given k, then
2. use the table via

```
algorithm FFS(x) begin
  if x = 0 then
    return 0
  else
    i ← 0
    while x ∧ (2^k − 1) = 0 do
      i ← i + k, x ← x ≫ k
    end
    return i + T[x ∧ (2^k − 1)]
  end
end
```

Notes:

- Solution #4: employ a divide-and-conquer style approach.

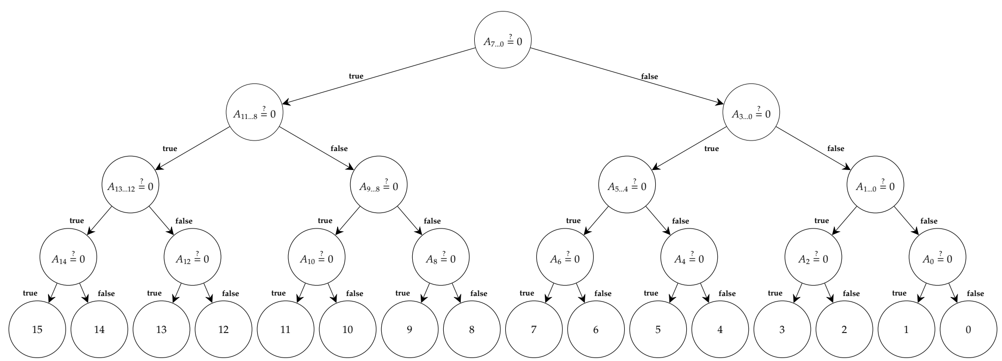
Algorithm

```
algorithm FFS(x) begin
  if x = 0 then
    return 0
  else
    i ← 1
    if x ∧ 0000FFFF(16) = 0 then  i ← i + 16, x ← x ≫ 16
    if x ∧ 000000FF(16) = 0 then  i ← i + 8, x ← x ≫ 8
    if x ∧ 0000000F(16) = 0 then  i ← i + 4, x ← x ≫ 4
    if x ∧ 00000003(16) = 0 then  i ← i + 2, x ← x ≫ 2
    if x ∧ 00000001(16) = 0 then  i ← i + 1, x ← x ≫ 1
    return i
  end
end
```


Notes:

- In English: we're basically using a binary decision tree, i.e.,

Algorithm



st. the challenge is to implement each step efficiently.

Notes:

Linux data structures and algorithms: bit-map (7)

Advanced operations: FFS

- In English: we're basically using a binary decision tree, i.e.,

Example

Given $w = 16$ (vs. $w = 32$) for simplicity, consider an

$$x = 1011110000000000_{(2)}$$

where

1. if x *is* non-zero it contains at least one 1, otherwise we early-abort,
2. we test whether
$$x \wedge 000000FF_{(16)} = 0$$
which is true since the 8 LSBs of x are 0,
3. this implies the 1 bit is within the 8 MSBs, so we update
$$i \leftarrow i + 8, x \leftarrow x \gg 8$$
i.e.,
   - update i st. it tracks the offset within x we are currently inspecting, then
   - discard the 8 LSBs of x.

st. the challenge is to implement each step efficiently.

Notes:

Linux data structures and algorithms: bit-map (8)

Advanced operations: FFS

Listing (linux-2.6.10/include/linux/arch/asm-arm/bitops.h)

```
#define fls(x) \
        ( __builtin_constant_p(x) ? generic_fls(x) : \
          ({ int __r; asm("clz\t%0, %1" : "=r"(__r) : "r"(x) : "cc"); 32-__r; }) )
#define ffs(x) ({ unsigned long __t = (x); fls(__t & -__t); })
```

Notes:

Linux data structures and algorithms: bit-map (9)

Advanced operations: FFS

Listing (`linux-2.6.10/include/linux/bitops.h`)

```c
static inline int generic_ffs(int x)
{
        int r = 1;

        if (!x)
                return 0;
        if (!(x & 0xffff)) {
                x >>= 16;
                r += 16;
        }
        if (!(x & 0xff)) {
                x >>= 8;
                r += 8;
        }
        if (!(x & 0xf)) {
                x >>= 4;
                r += 4;
        }
        if (!(x & 3)) {
                x >>= 2;
                r += 2;
        }
        if (!(x & 1)) {
                x >>= 1;
                r += 1;
        }
        return r;
}
```

Notes:

Linux data structures and algorithms: bit-map (10)

Advanced operations: POP

- Solution #1: utilise direct hardware support, e.g.,

  1. ARM ⇒ `vcnt`
  2. x86 ⇒ `popcnt`

Notes:

Linux data structures and algorithms: bit-map (10)

Advanced operations: POP

- **Solution #2**: perform direct, bit-by-bit process.

Algorithm

```
algorithm POP(x) begin
  if x = 0 then
    return 0
  else
    t ← 0
    for i = 0 upto w − 1 do
      if x ∧ 1 = 1 then
        t ← t + 1
      end
      x ← x ≫ 1
    end
    return t
  end
end
```

Notes:

Linux data structures and algorithms: bit-map (10)

Advanced operations: POP

- **Solution #3**: employ a pre-computation style approach.

Algorithm

1. first pre-compute a table st.
$$T[x] = \text{POP}(x)$$
for $0 \leq x < 2^k - 1$ given k, then
2. use the table via

```
algorithm POP(x) begin
  if x = 0 then
    return 0
  else
    t ← 0
    for i = 0 upto w/k − 1 do
      t ← t + T[x ∧ (2^k − 1)], x ← x ≫ k
    end
    return t
  end
end
```

Notes:

Linux data structures and algorithms: bit-map (10)

Advanced operations: POP

- Solution #4: employ a divide-and-conquer style approach.

Algorithm

algorithm POP(x) **begin**
  if $x = 0$ **then**
    return 0
  else
    $t \leftarrow 0$
    $x \leftarrow (x \wedge 55555555_{(16)}) + ((x \gg 1) \wedge 55555555_{(16)})$
    $x \leftarrow (x \wedge 33333333_{(16)}) + ((x \gg 2) \wedge 33333333_{(16)})$
    $x \leftarrow (x \wedge 0F0F0F0F_{(16)}) + ((x \gg 4) \wedge 0F0F0F0F_{(16)})$
    $x \leftarrow (x \wedge 00FF00FF_{(16)}) + ((x \gg 8) \wedge 00FF00FF_{(16)})$
    $x \leftarrow (x \wedge 0000FFFF_{(16)}) + ((x \gg 16) \wedge 0000FFFF_{(16)})$
    return t
  end
end

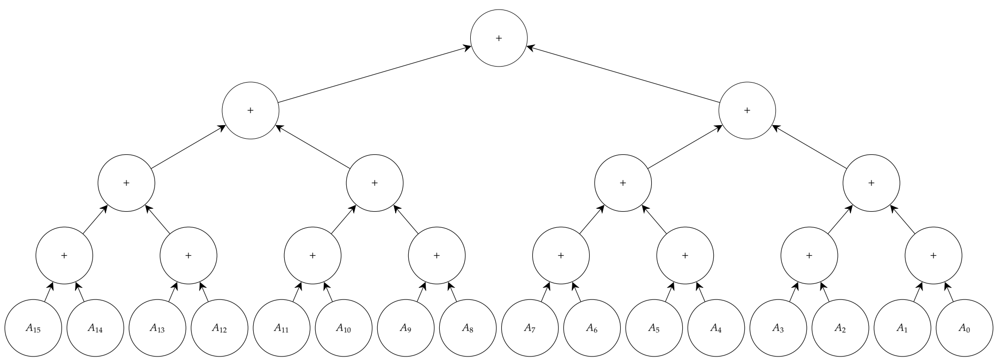
Notes:

Linux data structures and algorithms: bit-map (11)

Advanced operations: POP

- In English: we're basically using a binary addition tree, i.e.,

Algorithm



st. the challenge is to implement each step efficiently.

Notes:

Linux data structures and algorithms: bit-map (11)

Advanced operations: POP

- In English: we're basically using a binary addition tree, i.e.,

Example

Given $w = 16$ (vs. $w = 32$) for simplicity, consider an

$$x = 1011110001100001_{(2)}$$

where

1. if x *is* non-zero it contains at least one 1, otherwise we early-abort,
2. we compute
$$x \leftarrow (x \wedge 5555_{(16)}) + ((x \gg 1) \wedge 5555_{(16)}),$$
which is the sum of all 1-bit chunks,
3. this works because
$$\begin{array}{lcl c l} t_0 & = & x \wedge 5555_{(16)} & = & 0001010001000001_{(2)} \\ t_1 & = & (x \gg 1) \wedge 5555_{(16)} & = & 0101010000010000_{(2)} \end{array}$$
and hence $t_0 + t_1 = 0110100001010001_{(2)}$,
4. so, more generally, we
   - isolate the bits wrt. some chunk size (by masking then shifting them), then
   - add them together

   to implement each layer of the tree.

st. the challenge is to implement each step efficiently.

Notes:

Linux data structures and algorithms: bit-map (12)

Advanced operations: POP

Listing (`linux-2.6.10/include/linux/bitops.h`)

```
static inline unsigned int generic_hweight32(unsigned int w)
{
        unsigned int res = (w & 0x55555555) + ((w >> 1) & 0x55555555);
        res = (res & 0x33333333) + ((res >> 2) & 0x33333333);
        res = (res & 0x0F0F0F0F) + ((res >> 4) & 0x0F0F0F0F);
        res = (res & 0x00FF00FF) + ((res >> 8) & 0x00FF00FF);
        return (res & 0x0000FFFF) + ((res >> 16) & 0x0000FFFF);
}
```


Notes:

Conclusions

- Take away points:
  1. Good news: kernel data structures and algorithms $\simeq$ *general* data structures and algorithms.
     - use robust (abstract) complexity-driven analysis,
     - use robust (concrete) profile-driven analysis,
     - apply layered approach (i.e., multiple redundant data structures) if/when it makes sense,
     - apply the Pareto principle by focusing effort on on the dominant 80% not the 20%,
     - apply considered, well documented (vs. ad-hoc) optimisation techniques.

Notes:

Conclusions

- Take away points:

  2. Bad news: kernel data structures and algorithms $\neq$ *general* data structures and algorithms.
     - metrics used to evaluate

       - efficiency $\Rightarrow$ { kernel is pure overhead from user perspective, potentially even with real-time constraints
       - robustness $\Rightarrow$ any lack has *global* impact, meaning a crash is fatal!
       - maintainability $\Rightarrow$ { scale of kernel demands best practice wrt. good design and implementation

       present different challenges,
     - there are associated issues such as the lack of support available (the kernel cannot depend on the standard C library, for example),

Notes:

Additional Reading

- H.S. Warren Jr. *Hackers Delight*. Addison-Wesley, 2003.

Notes:

References

[1] *Wikipedia: Rob Pike*. URL: http://en.wikipedia.org/wiki/Rob_Pike (see pp. 6, 8, 10, 12, 14).

[2] H.S. Warren Jr. *Hackers Delight*. Addison-Wesley, 2003 (see p. 97).

[3] D.E. Knuth. *The Art of Computer Programming: Sorting and Searching*. 2nd ed. Vol. 3. 1997 (see p. 31).

[4] *Standard for Information Technology - Portable Operating System Interface (POSIX)*. Institute of Electrical and Electronics Engineers (IEEE) 1003.1-2008. 2008. URL: http://standards.ieee.org (see p. 60).

[5] C. Bays. "A Comparison of Next-fit, First-fit, and Best-fit". In: *Communications of the ACM (CACM)* 20.3 (1977), pp. 191–192 (see pp. 20, 22, 24, 26, 28).

[6] J. Bonwick. "The Slab Allocator: An Object-Caching Kernel Memory Allocator". In: *USENIX Summer Technical Conference (USTC)*. 1994, pp. 6–6 (see pp. 20, 22, 24, 26, 28).

[7] K.C. Knowlton. "A Fast Storage Allocator". In: *Communications of the ACM (CACM)* 8.10 (1965), pp. 623–625 (see pp. 20, 22, 24, 26, 28).

Notes: