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:

- Premise:
  - fundamentally, the kernel exists to support execution of (user mode) programs, so
  - better understanding of such programs leads to better understanding of the kernel.
- Goal: provide a recap on
  1. how programs are compiled, and
  2. some properties of programs during execution (as processes).

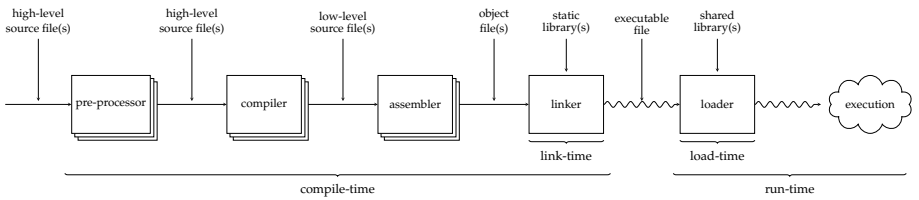
Notes:

- Keep in mind that program is used as an umbrella term here, capturing *any* software executed in user mode (under control of, and hence with support from the kernel): although they could differ in terms of their requirements and behaviour when considered in detail, for the sake of simplicity this will include application and system software. By considering how such programs are compiled and executed, the idea is we get a better understanding of a) requirements on and so the b) design and implementation plus c) behaviour of the kernel.

Compilation of programs (1)

Definition

A sequence of standard steps, implemented by a (user mode) tool-chain plus the kernel, translates a high-level program into a form ready to be executed:



Notes:

- The focus here is on compilation of C programs: alternatives introduce various complications, not least the distinction between needing to examine the program statically vs. dynamically (i.e., the need to do so at compile-time without considering execution, or using information only available at run-time).
  As a result, the C99 specification [3] is used as a *rough* guide wrt. terminology: although more modern versions of the standard exist, an openly accessible draft PDF for C99 exists that makes it easier to refer to. [2, Section 1.8] offers a somewhat similar introduction.
- Per [3, Section 5.1.1.1], the term **compilation unit** (or **translation unit**) describes the input to a compiler used to produce an associated **object file** as output. These terms are, in part, used to stress that a) each **source file** is pre-processed (so potentially includes some number of *other* such files) before compilation, and b) several object files are typically combined, by the linker, to form an **executable file**.
- It is important to distinguish between
  - a **static library** which is linked at link-time and hence included as part of the associated executable file, and
  - a **shared library** which is linked (dynamically) at load- or run-time and can hence be separate from but shared between multiple executable files.

  Some advantages of the latter include reduced size of executable files a) on-disk (since the library itself is not included), and b) in-memory (since only one instance of the library need be resident). However, notable disadvantages stem from the challenge of maintaining a compatible installation of such shared libraries: in Windows, where the term **Dynamic Link Library (DLL)** is used for the same concept, this problem is often described colloquially as "DLL hell".
- For format used for object and executable files needs to be richer than you might imagine: the latter, for example, does *not* consist of machine code instructions alone. Although alternatives exist, a UNIX-based tool-chain will typically use **Executable and Linkable Format (ELF)** [1] files in both cases. At a high level, it is important to keep in mind that such a file will contain
  1. a number of sections, each relating to content of a specific type; the sections roughly align with associated segments in the address space formed to support a process during execution, with examples including text and (initialised and uninitialised) data,
  2. a symbol table, each entry of which details a definition of some symbol, e.g., the identifier and address of a variable in the data section,
  3. a relocation table, each entry of which details a reference to some symbol: this guides the relocation process, detailing instructions that refer to a given symbol and so that must be rewritten iff. the address of said symbol is unknown or changes.
- Note that it is common to use the terms **program image** and **process image** to describe on-disk (static) and in-memory (dynamic) state: the former basically captures the executable (e.g., ELF) file, whereas the latter captures the address space of the associated process (having initially been populated from the executable file, and updated by execution).
- Although not shown explicitly in the diagram, an auxiliary input to the linker, namely a **linker script**, controls the linkage process: this essentially controls how the executable file is constructed, and, as a result, where entries are placed a) when merged together from multiple object files (e.g., if there are multiple text segments that need to be merged into one), and b) to reflect the layout expected by the kernel, and wrt. resources of the underlying platform (e.g., if the text segment should be placed in ROM or RAM).

Compilation of programs (2)

Definition

Within a given program, we might refer to an object via

- **abstract address** (or **symbolic address**), e.g., using an identifier

```
goto foo;
```

- **concrete address**, e.g., using a literal

```
uint32_t* bar = ( uint32_t* )( 0xDEADBEEF );
```

Definition

An abstract addresses must be **resolved** into a concrete address before execution; doing so effectively **binds** an identifier to the address at which an associated object is placed.

Definition

An object may be **relocated** (or moved) by the linker and/or loader, at link-time, load-time or run-time, implying the associated concrete address *changes*; all references to the relocated object (i.e., addresses as used by instructions) need to be rewritten to maintain correctness.

Notes:

- Although the terms are often used synonymously, an **identifier** is a name whereas a **symbol** captures a richer super-set of information: examples include the **type** of the object said identifier is naming.
- Within a compilation unit,
  - a **declaration** introduces a symbol *only*, implying the associated implementation exists elsewhere, whereas
  - a **definition** introduces a symbol *plus* associates it with an implementation: where the identifier is
    - a **variable**, the implementation implies allocation (and potentially initialisation) of memory, or
    - a **function**, the implementation implies the generation of low-level instructions that realise the high-level description

    and noting that the implementation is often referred to as an (or the associated) **object**,
  - a **reference** uses the previously introduced identifier for a symbol, in whatever way makes sense (i.e., wrt. the type and context).
- Per [3, Section 6.2.1], the **scope** of an identifier is a spatial concept in the sense it captures *where* the identifier can legally be referred to (i.e., used); an identifier may be described as **in scope** (resp. **out of scope**) if it can (resp. cannot) be referred to. Note that
  - **file scope** (or **global scope**) suggests the identifier can be referred to within all of the compilation unit, whereas
  - **block scope** (or **local scope**) suggests the identifier can only be referred to within part of the compilation unit (e.g., within a given function).

  Scopes are naturally hierarchical, st. the top-level file scope acts as a parent (or outer scope) to any child, block scope (or inner scope) it contains: in C, the body of a function defined within the file scope is such a case. They, in turn, act as parent to any sub-blocks (e.g., the body of a `for` loop). This nested hierarchy, demands rules to disambiguate cases where an identifier is shadowed (i.e., exists both in a parent and child scope simultaneously); C deems an more-inner scope to take precedence over a more-outer scope.
- Per [3, Section 6.2.1], a related question is whether the scope of a given identifier can extend beyond the compilation unit it is defined in: if so, this implies it is visible to the linker. The terms **external linkage** and **internal linkage** describe situations where an identifier is visible (resp. is hidden, or not visible) to the linker, and so can (resp. cannot) be referred to in another compilation unit. An identifier with global scope has external linkage by default; including the `static` storage-class keyword alters this to internal linkage.
- Per [3, Section 6.4.2], the **extent** of an identifier is a temporal concept in the sense it captures *when* the identifier can legally be refereed to; this is often termed the lifetime, or, more formally, storage duration of the associated object (i.e., the period for which storage is allocated for it: during this period is has a fixed address). C uses three classes:
  - **static extent** (or **global extent**) relates to identifiers with external or internal linkage, or no linkage and declared using the `static` storage-class keyword: the latter implies a variable has block scope, but an extent akin to one with file scope. The storage duration of a variable with static extent matches execution of the program, and, iff. it is initialised, this is performed once at load-time.
  - **automatic extent** relates to identifiers with no linkage and not declared using the `static` storage-class keyword: this implies a variable has block scope. The storage duration of a variable with automatic extent matches execution of the associated block, and, iff. it is initialised, this is performed every time said block is executed.
  - **allocated extent** relates to objects whose memory has been dynamic allocated via `malloc` or similar.

Compilation of programs (3)

- Example: consider a (contrived) program

Listing (`foo.c`)

```
1  #include   <stdio.h>
2  #include  <stdint.h>
3
4  #include <pthread.h>
5
6  extern uint64_t rdtsc();
7
8  static void* f( void* arg ) {
9    return NULL;
10 }
11
12 int main( int argc, char* argv[] ) {
13   pthread_t t;
14
15   int tsc_b = rdtsc();
16   pthread_create( &t, NULL, &f, NULL );
17   int tsc_a = rdtsc();
18
19   printf( "overhead = %d\n", tsc_a - tsc_b );
20
21   return 0;
22 }
```

Listing (`bar.s`)

```
1 .global rdtsc
2
3 rdtsc: rdtsc
4        salq   $32,  %rdx
5        orq    %rdx, %rax
6        ret
```

which aims to measure the overhead, in cycles, of thread creation ...

Notes:

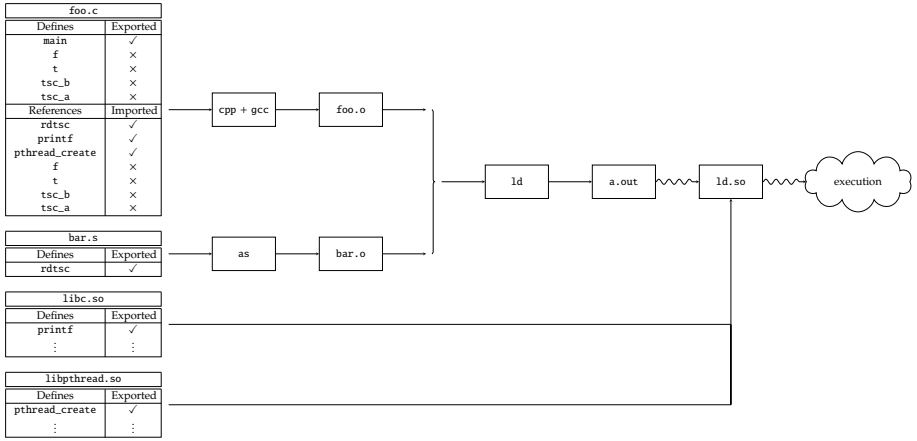
- We can attempt to describe various cases as follows

Declaration				
Type	Scope	Extent	Value	
Function	File			`int f( int x );`
Variable	File			`extern int x;`

Definition				
Type	Scope	Extent	Value	Example
Function	File	Static		`int f( int x ) { ... }`
Variable	File	Static	Initialised	`int x = 0;`
			Uninitialised	`int x;`
	Block	Static	Initialised	`static int x = 0;`
			Uninitialised	`static int x;`
	Block	Automatic	Initialised	`int x = 0;`
			Uninitialised	`int x;`
	Block	Allocated	Initialised	`int* x = ( int* )( malloc( 10 * sizeof( int ) ) );`

Compilation of programs (4)

- ... st. we could do dynamic linking



via

- `gcc -c -o foo.o foo.c`
- `as -c -o bar.o bar.s`
- `gcc -o a.out foo.o bar.o -lc -lpthread`

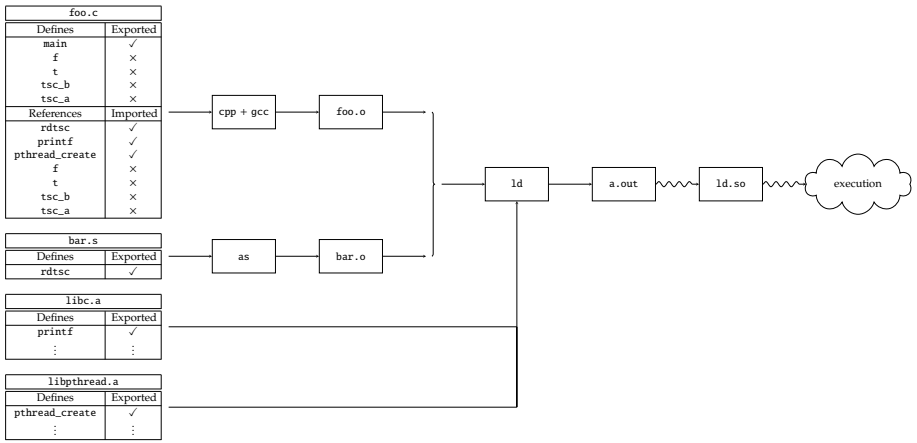
Notes:

- Although two options are outlined by the diagram, there are (at least) *three* possibilities for what and when linking is performed:
  - **Static linking** (or **early linking**) is performed by the (static) linker at link-time: a complete binding is said to exist, in the sense that any object referenced by an identifier will have a known address (st. the reference is resolved). Although static libraries represent the natural candidates for static linking, it *is* possible to perform static linking of shared libraries as well.
  - **Dynamic linking** (or **late linking**) is performed by the (dynamic) linker and/or loader at load-time; a partial binding is said to exist, in the sense that an object referenced by some identifier may have an unknown address (st. the reference is unresolved). The process is sometimes optimised using a pre-linking step, where aspects of the process normally be performed at load-time (thereby representing overhead) are pre-computed.
  - **Dynamic loading** represents a third option whereby a) the linking process is performed at run-time, and so even later than dynamic linking as described above, and b) may be invoked explicitly by the executing process, rather than implicitly by the kernel: on UNIX-like systems, libdl manages this process, e.g., via calls to `dlopen`, `dlsym` and `dlclose`.

  The third option is interesting, in so far as it enables "plug-in" style functionality to be implemented: the process can explore the set of additional components (i.e., object files) available, and decide on which it loads in order to extend or alter what it can do or does.
- One fair question is why `gcc` replaces `ld`, the linker in these examples. The answer is `gcc` automates the *entire* compilation process: *it* invokes `as` and `ld`, for example, for us if provided assembly language or object file(s) as input. To produce an executable file, the volume of command line arguments required for `ld` complicates matters so we opt to let `gcc` do it for us. Keep in mind this is simply a choice to reduce complexity, and that we *could* make use of `ld` manually if need be.
- It is useful to keep in mind various commands that can inspect various (intermediate) results produced within the example:
  - `objdump` and/or `readelf` can be used to inspect the content of object files (which, where ELF is used, is the same format for executables). For example, `objdump --syms foo.o` will dump the symbol table for the object file `foo.o`.
  - `ldd` can be used to list the shared libraries required by an executable file. For example, `ldd a.out` will do so for the executable file `a.out`: in the first dynamically linked case the shared libraries `libc.so` and `libpthread.so` are listed, but then in the second statically linked case they are not.

Compilation of programs (4)

- ... st. we could do static linking



via

- `gcc -c -o foo.o foo.c`
- `as -c -o bar.o bar.s`
- `gcc -static -o a.out foo.o bar.o -lc -lpthread`

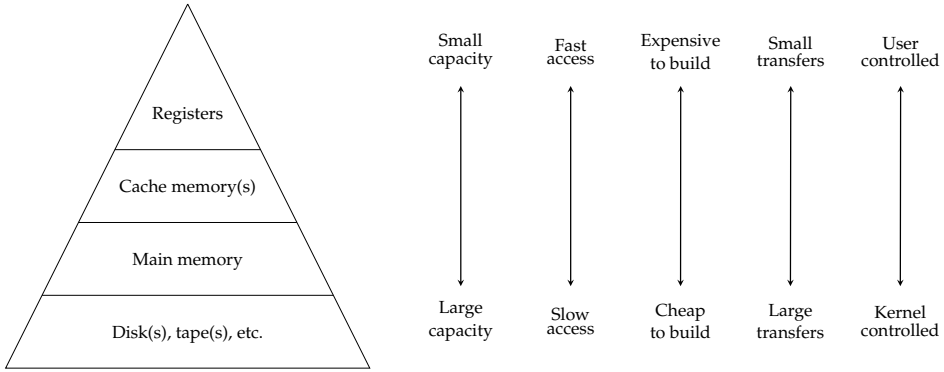
Notes:

- Although two options are outlined by the diagram, there are (at least) *three* possibilities for what and when linking is performed:
  - **Static linking** (or **early linking**) is performed by the (static) linker at link-time: a complete binding is said to exist, in the sense that any object referenced by an identifier will have a known address (st. the reference is resolved). Although static libraries represent the natural candidates for static linking, it *is* possible to perform static linking of shared libraries as well.
  - **Dynamic linking** (or **late linking**) is performed by the (dynamic) linker and/or loader at load-time; a partial binding is said to exist, in the sense that an object referenced by some identifier may have an unknown address (st. the reference is unresolved). The process is sometimes optimised using a pre-linking step, where aspects of the process normally be performed at load-time (thereby representing overhead) are pre-computed.
  - **Dynamic loading** represents a third option whereby a) the linking process is performed at run-time, and so even later than dynamic linking as described above, and b) may be invoked explicitly by the executing process, rather than implicitly by the kernel: on UNIX-like systems, libdl manages this process, e.g., via calls to `dlopen`, `dlsym` and `dlclose`.

  The third option is interesting, in so far as it enables "plug-in" style functionality to be implemented: the process can explore the set of additional components (i.e., object files) available, and decide on which it loads in order to extend or alter what it can do or does.
- One fair question is why `gcc` replaces `ld`, the linker in these examples. The answer is `gcc` automates the *entire* compilation process: *it* invokes `as` and `ld`, for example, for us if provided assembly language or object file(s) as input. To produce an executable file, the volume of command line arguments required for `ld` complicates matters so we opt to let `gcc` do it for us. Keep in mind this is simply a choice to reduce complexity, and that we *could* make use of `ld` manually if need be.
- It is useful to keep in mind various commands that can inspect various (intermediate) results produced within the example:
  - `objdump` and/or `readelf` can be used to inspect the content of object files (which, where ELF is used, is the same format for executables). For example, `objdump --syms foo.o` will dump the symbol table for the object file `foo.o`.
  - `ldd` can be used to list the shared libraries required by an executable file. For example, `ldd a.out` will do so for the executable file `a.out`: in the first dynamically linked case the shared libraries `libc.so` and `libpthread.so` are listed, but then in the second statically linked case they are not.

Execution of programs (1)

Definition



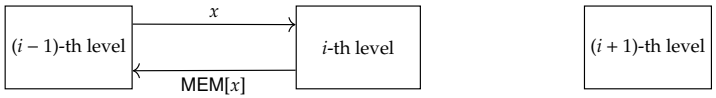
Notes:

- Some important (by design) implications stem from this organisation:
  1. The general principle is that a) an access to data held in a higher level can be satisfied more efficiently than a lower level, *but* b) higher levels have smaller capacity, so can only retain a subset of the content held in lower levels.
  2. As a result, various mechanisms, operating at the interfaces between levels, a) move data from the lower to a higher level in order to make access to the working set as efficient as possible, while also b) evict and so move data from the higher to a lower level to mitigate their mismatched capacity (i.e., create free space where necessary).
  3. Some levels (e.g., the cache) are managed (semi-)automatically by the underlying hardware, whereas others are not: the kernel is responsible for management of the lower levels, meaning it must allocate space on and interact explicitly with the disk.
  4. If phrased within the context of (i.e., at interface between) cache and memory, the description above perhaps already makes sense. Crucially, however, it is important to keep in mind that largely similar mechanisms manage the other interfaces: one can identify a) register allocation and spilling (as implemented by the compiler) at the interface between registers and cache, b) demand paged virtual memory (or perhaps swapping even more generally, either way implemented by the kernel) at the interface between memory and disk. That is, when considering the complexities of demand paging, remember that in essence it simply treats memory wrt. disk in the same way you already know the processor treats cache wrt. memory. More concretely, issues such as replacement policy are basically the same even if differing trade-offs suggests a different choice of solution in each case.
- [4, Section 3] uses (and probably introduces) the term **von Neumann bottleneck** in a very general context: the discussion focuses on how the von Neumann architecture acts as a general limit wrt. programs and programming. A more specific interpretation of the same argument focuses on the inherent limit on throughput between a processor and memory. In particular, since executing an instruction requires fetching it from memory, the latency of memory access bounds efficiency of instruction execution. Since advances in technology mean the former has improved more slowly than the latter, a given processor is therefore limited by the resulting bottleneck; this has been termed the **memory wall** [8].
  Widening this bottleneck, i.e., improving throughput of memory access, is a clear motivation for the memory hierarchy existing. For example, use of caches, or separate, Harvard-esque, paths for instruction and data access, are both mechanisms to do so.

Execution of programs (2)

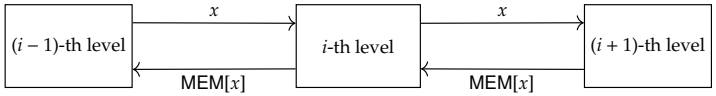
Definition

Imagine an access to address x is made by the $(i-1)$-th level: if the i-th level *does* hold the associated data (i.e., x *is* **resident**), a **hit** occurs and the data is available quickly, without accessing the $(i+1)$-th level:



Definition

Imagine an access to address x is made by the $(i-1)$-th level: if the i-th level does *not* hold the associated data (i.e., x is *not* **resident**), a **miss** occurs and the data is available only after a subsequent, slower access to the $(i+1)$-th level:



In this case, the i-th level (potentially) retains the data; it may need to **evict** data (i.e., via a **replacement policy**) that it already holds in order to make space.

Notes:

Execution of programs (3)

Definition (Hill and Smith [6, Section V.A])

A miss occurs at the i-th level due to the non-residency of some address when accessed: the underlying *reason* is often classified by terming it a

1. a **cold miss** due to the i-th level being initially empty wrt. the working set,
2. a **capacity miss** due to the i-th level having too small a capacity to hold the entire working set of addresses accessed,
3. a **conflict miss** due to the non-injective mapping of addresses from the $(i+1)$-th level to those in the i-th level.

Notes:

- The description of a conflict miss is quite terse: it is essentially due to multiple addresses in the larger, higher level mapping to and thus competing for the same address in the smaller, lower level. It is common to term this **interference**, which stems from various underlying sources (e.g., see [7]).

Execution of programs (4)

Definition

During execution of an (average) program,

- accesses exhibit **spatial locality** if address x being accessed at time t, implies address $x \pm \delta$ is likely to be accessed at time $t+1$, whereas
- accesses exhibit **temporal locality** if address x being accessed at time t implies address x is likely to be accessed at time $t+1$.

Notes:

- Written in English,
  - **spatial locality** is the trend that having made an access to one address, it is probable that the next access (within a small window of time) will be to the some address close to the first whereas
  - **temporal locality** is the trend that having made an access to one address, it is probable that the next access (within a small window of time) will be to the same address as the first.

Example

Consider the following C function which increments each element in an array `A`:

```
1 void inc( float* A, int n ) {
2   for( int i = 0; i < n; i++ ) {
3     A[ i ] = A[ i ] + 1;
4   }
5 }
```

- Accesses to elements in the array are **spatially local**: having accessed the `i`-th element it is probable the (`i` + 1)-th element will be accessed next.
- The fetch of the addition instruction within the loop body is **temporally local**: having fetched it once, only one time in n do we *not* fetch it again (i.e., when the loop terminates).

Notes:

Definition (Denning [5, Section 3])

Imagine a process $\mathbf{P}_i$ is executing:

$$\mathcal{W}_{t,\delta}(\mathbf{P}_i)$$

denotes the associated **working set**, i.e., the set of addresses (or portion of address space) in use during the time period between $t - \delta$ and t; it may be convenient to omit either t and/or δ, e.g., to imply interest in the "current" time.

Notes:

- In terms of defining what working set means, "current" can be a tricky concept to capture. Use of δ approximates it using window of time around t: if the principle of locality applies, the intuition is that those addresses used recently *should* approximate those "currently" useful and so "in use".
- Denning [5, Section 3] describes the working set as "the smallest collection of information that must be present in main memory to assure efficient execution". At least two (subtle) points are with keeping in mind:
  1. A reasonable interpretation of "in use" is that $\mathbf{P}_i$ has used (e.g., fetched an instruction or either stored data at or loaded data from) an address during the specified period of time. This ignores issues such as access frequency however, since it is also reasonable to assume an some addresses will be are more important than others.
     Consider addresses x and y accessed once at time $t = 0$, and at times $t \in \{1, 2, \ldots 99\}$ respectively. Within a window of $\delta = 100$ time units *both* are access, and thus sit within the working set of $\mathbf{P}_i$. However, it it seems reasonable to assume that y is more likely to cause inefficiency vs. x if accesses to it are inefficient.
  2. Although couched in terms related to the memory hierarchy, it is reasonable to generalise the concept: fundamentally, it is concerned with resource allocation and that, at a given point in time, a process has an immediate requirement for a subset of the available resources.

Execution of programs (7)

Definition

A process is said to be *x*-**bound** if *x* limits how quickly it can execution: it will be

- **CPU-bound** (or **compute-bound**) if the time it takes to complete is dominated by the instruction execution rate of the processor, or
- **I/O-bound** if the time it takes to complete is dominated by the time waiting for I/O operations to complete

noting that **memory-bound** processes are a sub-class of I/O-bound processes relating specifically to memory access.

Definition

The terms *x*-**burst** and *x*-**wait** describe periods of time when a process is either doing *x* or waiting for *x* respectively. Note that

- CPU-bound processes typically have *long* CPU-bursts and *few* I/O-waits, while
- I/O-bound processes typically have *short* CPU-bursts and *many* I/O-waits.

Notes:

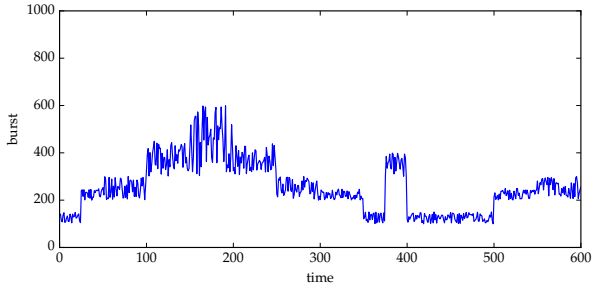
- In rough terms you could think of compute- and I/O-bound processes as those which *mostly* do computation (which is local to the processor) or I/O (which involves devices remote from the processor).
  This is too simplistic in some cases, however, because a process might not do much I/O yet still be I/O-bound (e.g., if the associated device is slow relative to the processor). As such, whatever *bottleneck* limits (or bounds) the process will be important: if the process is *x*-bound for whatever *x* is, then it should execute faster if we could make *x* less of a bottleneck.
- A *typical* process will alternate between CPU-bursts and I/O-waiting, in a kind of cycle: this is obvious, if you consider that such processes will typically accept input, compute some output, then produce that output.
- The fact that a process may need to wait for an I/O operation to complete, even if technically this is a one-off vs. long-term limit, offers a clear motivation for supporting the concept of multi-tasking.
  Imagine each process spends some fraction of time c doing computation. One might argue that having $n = 100/c$ processes would yield perfect, 100% utilisation of the processor due to multi-tasking (vs. much less if multi-tasking were *not* used): there would always be one process able to make progress if it were scheduled for execution. Even if there *were* n such processes available to support this fact, it is too simple a model because, in reality, however, each process also spends a fraction b blocked waiting for access to or for the I/O device itself. Therefore, the probability that *all* processes are blocked is b^n and so the processor is likely idle for $1 - b^n$ of the time.

Execution of programs (8)

- Problem: the future behaviour of $\mathbf{P}_i$ is unknown wrt. burst and/or wait period.
- Solution: *predict* it based on previous behaviour, e.g.,
  1. let
     R_t denote the measured burst period at time t
     S_t denote the predicted burst period at time t
  2. update the prediction via a moving average, i.e.,
     $$S_{t+1} = (1 - \alpha) \cdot S_t + \alpha \cdot R_t$$
     where $\alpha = 0.5$ say.

Notes:

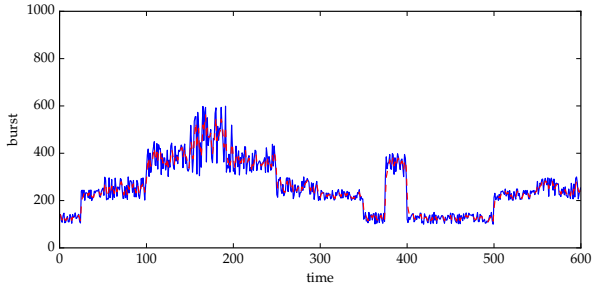
Example



Notes:

Execution of programs (9)

Example



Notes:

Conclusions

- Take away points:
  1. Understanding program compilation is *vital*:
     - the compilation tool-chain needs to adhear to an interface defined by the kernel,
     - the kernel itself *is* a program.
  2. Understanding program execution is *vital*:
     - supports robust assumptions,
     - will (partially) dictate the interface required between program and kernel,
     - will (partially) dictate how the kernel allocates and manages resources

  noting purely (static) assumption can be supplemented by (dynamic) profiling modulo the overhead.

Notes:

References

[1] *Wikipedia: Executable and Linkable Format*. URL: http://en.wikipedia.org/wiki/Executable_and_Linkable_Format (see p. 8).

[2] A.S. Tanenbaum and H. Bos. *Modern Operating Systems*. 4th ed. Pearson, 2015 (see p. 8).

[3] *Information technology – Programming languages – C*. Internation Organization for Standardization (ISO) Standard 9899:1999. 1999. URL: http://www.iso.org (see pp. 8, 10).

[4] J. Backus. "Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs". In: *Communications of the ACM (CACM)* 21.8 (1978), pp. 613–641 (see p. 18).

[5] P.J. Denning. "The Working Set Model for Program Behavior". In: *Communications of the ACM (CACM)* 11.5 (1968), pp. 323–333 (see pp. 27, 28).

[6] M.D. Hill and A.J. Smith. "Evaluating associativity in CPU caches". In: *IEEE Transactions on Computers* 38.12 (1989), pp. 1612–1630 (see p. 21).

[7] O. Temam, C. Fricker, and W. Jalby. "Cache interference phenomena". In: *ACM SIGMETRICS Performance Evaluation Review* 22.1 (1994), pp. 261–271 (see p. 22).

[8] W.A. Wulf and S.A. McKee. "Hitting the memory wall: implications of the obvious". In: *ACM SIGARCH Computer Architecture News* 23.1 (1995), pp. 20–24 (see p. 18).

Notes: