Concurrent Computing (Operating Systems)

Daniel Page

Department of Computer Science,
University Of Bristol,
Merchant Venturers Building,
Woodland Road,
Bristol, BS8 1UB. UK.
⟨csdsp@bristol.ac.uk⟩

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:

- Problem:
  - Imagine we want to communicate an n-byte buffer x from process $\mathbf{P}_i$ to process $\mathbf{P}_j$.
  - As described so far, the processes
    - are *totally* protected wrt. each other,
    - can be executed in *any* order,
    - can be suspended at *any* time,
    - can be remain suspended for *any* period, and
    - can be resumed at *any* time.
- Solution: we need mechanisms to
  1. identify the end-points,
  2. synchronise the end-points, and
  3. communicate the data,

  i.e., **Inter-Process Communication (IPC)**.

Notes:

- In short, the goal in designing and implementing an IPC mechanism is to relax constraints on protection st. communication between processes is possible. Ideally said mechanism supports synchronisation, because it is often a *requirement* for correctness, but maximises concurrency (and hence utilisation): there is no point being *so* aggressive wrt. synchronisation that the advantages of multi-programming are prevented!

- Note that:
  - most UNIX-like kernels support *several* IPC interfaces, but we'll focus on POSIX only,
  - there are *numerous* alternatives summarised by

Type	Standard	Mechanism	Identifier
Synchronisation	System V	semaphore	keyed
	POSIX	semaphore	named or unnamed
Notification	POSIX	signal	PID
Communication	System V	shared memory	keyed
	POSIX	shared memory	named
	Linux	shared mapping	named or unnamed
	System V	message queue	keyed
	POSIX	message queue	named
	POSIX	pipe	named or unnamed
	POSIX	domain socket	named or unnamed

  but we'll focus on a few only.

Notes:

- We devote limited coverage to (at least) three aspects which are, none the less, still important. These are:
  1. Any good IPC interface will include a way to communicate errors from the kernel to user space. There are, of course, many ways to achieve this: return codes and use of the C standard library global variable `errno` are two examples.
  2. A given IPC mechanism can described in terms of **persistence**, meaning the period of time is can be used or, respectively, the period of time any resources supporting it persist. For example, consider that a given mechanism may be
     - *process* persistent if the resource exists until the last process referencing it (i.e., holds an open descriptor) either dealloates it, closes it or terminates (st it is implicitly deallocated or "cleaned-up"),
     - *kernel* persistent if the resource exists until explicitly deallocated, or the kernel is rebooted, or
     - *file system* persistent if the resource exists until explicitly deallocated.
  3. Where any resource supporting an IPC mechanism is named, the name may exist within a **name space**. This might be dedicated, and so disjoint from any *other* name space st. name collisions are avoided; it may demand a specific format by convention (e.g., per a file system path), or may be free form.
  4. A given IPC mechanism typically includes a concept of controlled **protection**, which applies to the descriptor a process holds and uses wrt. said mechanism. For example, POSIX shared memory (much like other POSIX-based IPC mechanisms) allows specification of this property via flags when invoking `shm_open`: `O_RDONLY` opens the shared memory resource for only read access, whereas `O_RDWR` opens it for both read and write access.
- For System V cases:
  - The scheme for selecting function identifiers tends to follow an `XY` format (e.g., `semget`).
  - The term keyed refers to specification of the resource using a key of type `key_t`; typically this is will be generated from a name (i.e., a string) using the function `ftok` to hash the latter into the former.
- For POSIX cases:
  - The scheme for selecting function identifiers tends to follow an `X_Y` format (e.g., `sem_open`).
  - The term named refers to specification of the resource using a name (i.e., a string); typically a this formatted to match some convention, meaning the name is essentially file name within a virtual file system. The corresponding unnamed case is normally termed anonymous, but we stick with the former for consistency.
  - The functions that allocate named resources typically return a descriptor, or handle, which is compatible with a standard file descriptor; this makes sense, because it refers to a resource supported by a virtual file system. A duality results, meaning, for example, that one can resize a POSIX shared memory resource using `ftruncate` or close it with `close`.
  - When a virtual file system is used to support resources, this is typically exposed to the user: POSIX shared memory resources can be inspected at `/dev/shm`, for example.
  - When developing programs that use some APIs, the POSIX real-time library `librt` must be included in the link process: for example, one might include `-lrt` as an option to GCC.

IPC-related synchronisation: semaphore (1)

Definition

A **critical region** (or **critical section**) is a portion of a (multi-threaded) program that may not be executed concurrently (i.e., not executed by more than one thread of execution at the same time). A typical example is access to some shared resource, which, if it *were* concurrent, would fail somehow.

Notes:

- Assuming we want user space semaphores, one could imagine (at least) two implementation options, namely
  - unsupported, user space:
    - \+ doesn't need a system call to enter kernel mode,
      needs care wrt. atomicity, but often supported by processor (e.g., test-and-set),
    - – does need shared memory between processes,
    - – a process blocked waiting for access must spin
  - supported, kernel/user space:
    - – does need a system call to enter kernel mode,
      needs care wrt. atomicity, but only within kernel,
    - \+ doesn't need shared memory between processes,
    - \+ a process blocked waiting for access can be suspended

  plus some intermediate options.
- The identifiers used for the two operations is historical: they stem from Dutch words used by Dijkstra [19]. It is reasonable to think of V as meaning release (or signal, or sometimes "up" to align with increment), and P as meaning acquire (or wait, or sometimes "down" to align with decrement).
- The bracket notation here captures the idea of atomicity: writing $[x]$ means the instructions that implement x must be executed atomically, st. they cannot be interleaved with *other* statements (e.g., due to a context switch from one process to another). Intuitively, one can think of x as, from an external perspective such as another process, "looking like" a single rather than multiple instructions. Note that the term *instruction* is used carefully. It does *not* follow that if x is a single *statement*, e.g., a C statement such as an assignment, it will translate into a single instruction. That is, you do *not* get atomicity "for free" by using a single statement.
  It is easy to think of cases to motivate this restriction. Consider V, for example, where we must load s, increment it and then store it. If these steps were not atomic, then it *could* the case that some other process updates s between it being loaded and stored by the first: this essentially reverts the value of s, overwriting the update performed by the other process.
- Thinking about atomicity a little more deeply leads to a relationship, or duality between, it and concurrency. More specifically, it is reasonable to think as follows:
  - concurrency is the illusion that multiple instructions occur in parallel, motivated by utilisation and hence efficiency; to manufacture the illusion, we must suspend and resume those instructions (e.g., because there is one processor which can execute them), whereas
  - atomicity is the illusion that multiple instructions are one instruction (i.e., occurred at once), motivated by correctness; to manufacture the illusion, we must *not* suspend and resume those instructions (i.e., force their completion).

  As such, the two concepts are (up to a point) mutually exclusive: in the former case we want to maximise concurrency whereas in the latter we want to minimise it (i.e., sequentialise, or prevent concurrency to ensure the atomic execution property). Clearly this suggests a compromise so we can achieve both where need be; to some extent the scheduler is responsible for making this compromise, depending on the (dynamic) requirements of executing processes.
- Realising atomicity can be challenging, so hardware support is normally provided in the shape of special-purpose instructions. For example:

IPC-related synchronisation: semaphore (1)

Definition (Dijkstra [19])

A **semaphore** s is a counter, equipped with two operations

$$
\begin{array}{lcl}
V(s,x) & := & [\ s \leftarrow s + x\ ] \\
P(s,x) & := & \textbf{forever do}\ [\ \textbf{if}\ s \geq x\ \textbf{then}\ s \leftarrow s - x,\ \textbf{break}\ ]
\end{array}
$$

to increment and decrement it by x (typically $x = 1$). The semaphore is used to control concurrent access to some resource: intuitively, s is the number of concurrent users allowed (resp. "units" of the resource available) and P waits until access is allowed.

Definition

A **mutex** can be described as a *binary* semaphore (cf. a counting semaphore, a generalisation from 2 to n values) that simply allows or disallows access.

Notes:

- Assuming we want user space semaphores, one could imagine (at least) two implementation options, namely
  - unsupported, user space:
    - \+ doesn't need a system call to enter kernel mode,
      needs care wrt. atomicity, but often supported by processor (e.g., test-and-set),
    - – does need shared memory between processes,
    - – a process blocked waiting for access must spin
  - supported, kernel/user space:
    - – does need a system call to enter kernel mode,
      needs care wrt. atomicity, but only within kernel,
    - \+ doesn't need shared memory between processes,
    - \+ a process blocked waiting for access can be suspended

  plus some intermediate options.
- The identifiers used for the two operations is historical: they stem from Dutch words used by Dijkstra [19]. It is reasonable to think of V as meaning release (or signal, or sometimes "up" to align with increment), and P as meaning acquire (or wait, or sometimes "down" to align with decrement).
- The bracket notation here captures the idea of atomicity: writing $[x]$ means the instructions that implement x must be executed atomically, st. they cannot be interleaved with *other* statements (e.g., due to a context switch from one process to another). Intuitively, one can think of x as, from an external perspective such as another process, "looking like" a single rather than multiple instructions. Note that the term *instruction* is used carefully. It does *not* follow that if x is a single *statement*, e.g., a C statement such as an assignment, it will translate into a single instruction. That is, you do *not* get atomicity "for free" by using a single statement.
  It is easy to think of cases to motivate this restriction. Consider V, for example, where we must load s, increment it and then store it. If these steps were not atomic, then it *could* the case that some other process updates s between it being loaded and stored by the first: this essentially reverts the value of s, overwriting the update performed by the other process.
- Thinking about atomicity a little more deeply leads to a relationship, or duality between, it and concurrency. More specifically, it is reasonable to think as follows:
  - concurrency is the illusion that multiple instructions occur in parallel, motivated by utilisation and hence efficiency; to manufacture the illusion, we must suspend and resume those instructions (e.g., because there is one processor which can execute them), whereas
  - atomicity is the illusion that multiple instructions are one instruction (i.e., occurred at once), motivated by correctness; to manufacture the illusion, we must *not* suspend and resume those instructions (i.e., force their completion).

  As such, the two concepts are (up to a point) mutually exclusive: in the former case we want to maximise concurrency whereas in the latter we want to minimise it (i.e., sequentialise, or prevent concurrency to ensure the atomic execution property). Clearly this suggests a compromise so we can achieve both where need be; to some extent the scheduler is responsible for making this compromise, depending on the (dynamic) requirements of executing processes.
- Realising atomicity can be challenging, so hardware support is normally provided in the shape of special-purpose instructions. For example:

- ARMv7-A provides two forms of support in hardware, namely
  1. atomic load and store [18, Section 1.2.1]:

     `ldrex r1, [ r0 ]` $\mapsto \begin{cases} \mathsf{GPR}[1] \leftarrow \mathsf{MEM}[\mathsf{GPR}[0]] \\ \text{update monitor(s) wrt. address } x = \mathsf{GPR}[0] \end{cases}$

     `strex r2, r1, [ r0 ]` $\mapsto$
     > **if** *monitor(s) allow access to address* $x = \mathsf{GPR}[0]$ **then**
     > | $\mathsf{MEM}[\mathsf{GPR}[0]] \leftarrow \mathsf{GPR}[1]$
     > | $\mathsf{GPR}[2] \leftarrow 0$
     > **else**
     > | $\mathsf{GPR}[2] \leftarrow 1$
     > **end**

  2. atomic swap [18, Appendix A]:

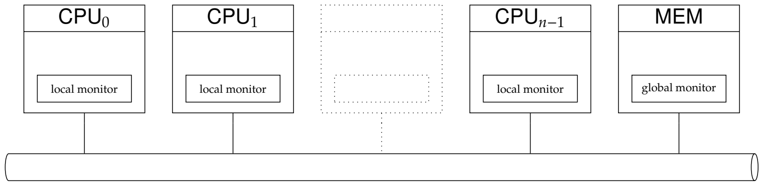
     `swp r2, r1, [ r0 ]` $\mapsto \begin{cases} t \leftarrow \mathsf{MEM}[\mathsf{GPR}[0]] \\ \mathsf{MEM}[\mathsf{GPR}[0]] \leftarrow \mathsf{GPR}[2] \\ \mathsf{GPR}[1] \leftarrow t \end{cases}$

  the former of which is now (strongly) preferred.

Notes:

- As with other load and store instructions, a suite of byte, half-word and double-word variants are also provided (e.g., `ldrexb` and `strexb`). The behaviour of these is equivalent in the context of exclusive access (only the access granularity differs), so we ignore then from here on.

- Idea: hardware-based **exclusive access monitors**.

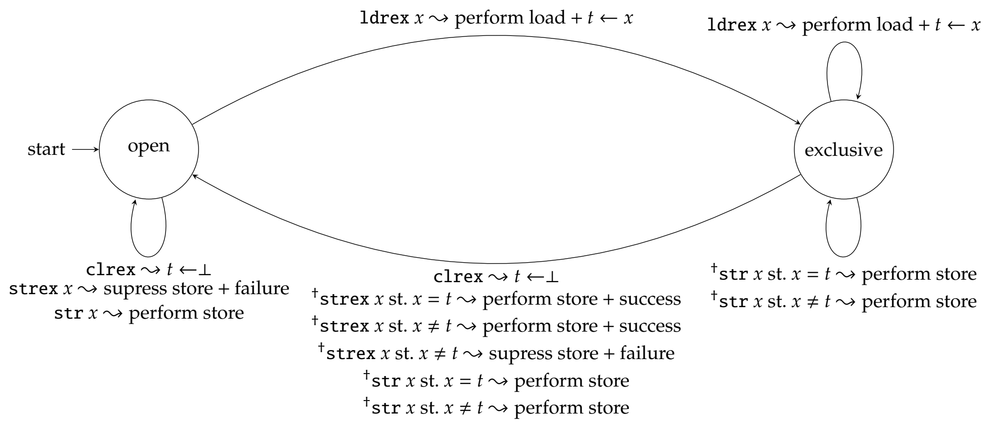


  - two types, namely local and global, of monitor are operated; these are essentially state machines,
  - for a given access to $\mathsf{MEM}[x]$,
    1. non-shared region $\Rightarrow$ checked wrt. local *only*
    2. shared region $\Rightarrow$ checked wrt. local *plus* global monitor
  - `ldrex` from address x succeeds, but updates the monitor(s) by "tagging" (or remembering) x,
  - `strex` to address x succeeds iff. there was no more recent store wrt. x, otherwise need to retry.

Notes:

- Design of this mechanism is motivated (vs. `swp`) by the need to scale, and, in particular, support multi-processor systems.
- Intuitively, what this mechanism is doing is decoupling the two halves of what might normally be an atomic instruction, i.e., loading and updating a value in memory: this is most obvious when comparing `swp` with a *pair* of `ldrex` and `strex`. As a result, the mechanism is general-purpose in the sense it allows *any* operation between the pair (vs. a fixed operation, e.g., with `swp`).
  Provided the gap between `ldrex` and `strex` is small ([17, Section A3.4.5] recommends 128B at most), the latter will succeed whp. without a need to retry (i.e., without an overhead wrt. instruction latency) *but* always fail if exclusive access is violated.
- The micro-architectural location of global monitor can differ, but remains easiest to consider it at the (shared, global) memory interface; this sort of mirrors the role, as a means of enforcing consistency wrt. the memory content as accessed by the n processors.
- A tag for address x is the l MSBs, i.e., $t = \mathsf{MSB}_l(x)$, where l is implementation defined: this is termed the **Exclusives Reservation Granule (ERG)**, and essentially defines how large each tagged *region* is. Typical values range from 8B to 2048B, so care is needed to prevent false positives (e.g., allowing two semaphores to occupy the same ERG-based region, and thereby aliasing each other wrt. exclusive access).

Algorithm ([18, Figure 1-1] and [17, Figure A3-3])



Notes:

- Cases marked with † are implementation defined, which is to say they may, legitimately, differ on different implementations of ARMv7-A. Description of a global monitor is *significantly* more complicated than a local monitor, but keep in mind that essentially the *same* structure and reasoning applies

IPC-related synchronisation: semaphore (4)
ARMv7-A synchronisation primitives

Algorithm ([18, Figure 1-1] and [17, Figure A3-3])

Q	Operation	Effect	Q'	
open	clrex	update $t \leftarrow \perp$	open	
	ldrex x	perform load, update $t \leftarrow x$	exclusive	
	strex x	supress store, return failure	open	
	str x	perform store	open	
exclusive	clrex	update $t \leftarrow \perp$	open	
	ldrex x	perform load, update $t \leftarrow x$	exclusive	
	strex x st. $x = t$	perform store, return success	open	
	strex x st. $x \neq t$	perform store, return success supress store, return failure	open	† †
	str x st. $x = t$	perform store	exclusive open	† †
	str x st. $x \neq t$	perform store	exclusive open	† †

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

Notes:

- Cases marked with † are implementation defined, which is to say they may, legitimately, differ on different implementations of ARMv7-A. Description of a global monitor is *significantly* more complicated than a local monitor, but keep in mind that essentially the *same* structure and reasoning applies

Example

user space		hardware				
$\mathbf{P}_i$	$\mathbf{P}_j$	Q	t	Q'	t'	effect
`ldrex` x		open	$\perp$	exclusive	x	perform load
`strex` x		exclusive	x	open	x	perform store, return success
	`ldrex` x	open	x	exclusive	x	perform load
	`strex` x	exclusive	x	open	x	perform store, return success

time

Notes:

- This example is simplistic, in the sense it a) relates to a local monitor only, but also b) involves two processes:
  - The x shown here and elsewhere is a *physical* address: the example is simple because virtual memory is ignored, and thus, in reality, $\mathbf{P}_i$ and $\mathbf{P}_j$ will have separate virtual address spaces.
  - [17, Section A3.4.4] describes that the kernel is required to reset the local monitor during a context switch: this should be read as switch of virtual address spaces, and means the execution of a `clrex` instruction. Basically this is a way of preventing false positives (i.e., a virtual address x from one process matching a tagged virtual address x' from some other process); the argument is similar to the reason we need to flush the TLB during a context switch. However, the example then fails to work: if we reset Q to open after each context switch, the failure case shown never occurs!
  - Indeed this is a problem, but one solved by adding some detail to the example. If virtual memory *was* considered after all, notice that the only way the processes could make accesses to the same, physical x is if that address was within a shared region. In such cases the global monitor would be used as a second check, and this detects an inconsistency so causes the store to fail as expected.

Example

user space		hardware				
$\mathbf{P}_i$	$\mathbf{P}_j$	Q	t	Q'	t'	effect
`ldrex` x		open	$\perp$	exclusive	x	perform load
	`ldrex` x	exclusive	x	exclusive	x	perform load
`strex` x		exclusive	x	open	x	perform store, return success
	`strex` x	open	x	open		supress store, return failure

time

Notes:

- This example is simplistic, in the sense it a) relates to a local monitor only, but also b) involves two processes:
  - The x shown here and elsewhere is a *physical* address: the example is simple because virtual memory is ignored, and thus, in reality, $\mathbf{P}_i$ and $\mathbf{P}_j$ will have separate virtual address spaces.
  - [17, Section A3.4.4] describes that the kernel is required to reset the local monitor during a context switch: this should be read as switch of virtual address spaces, and means the execution of a `clrex` instruction. Basically this is a way of preventing false positives (i.e., a virtual address x from one process matching a tagged virtual address x' from some other process); the argument is similar to the reason we need to flush the TLB during a context switch. However, the example then fails to work: if we reset Q to open after each context switch, the failure case shown never occurs!
  - Indeed this is a problem, but one solved by adding some detail to the example. If virtual memory *was* considered after all, notice that the only way the processes could make accesses to the same, physical x is if that address was within a shared region. In such cases the global monitor would be used as a second check, and this detects an inconsistency so causes the store to fail as expected.

Listing ([18, Example 1-6])

```
sem_post: ldrex     r1, [ r0 ] ; s' = MEM[ &s ]
          add   r1, r1, #1     ; s' = s' + 1
          strex r2, r1, [ r0 ] ; r <= MEM[ &s ] = s'
          cmp   r2, #0         ;     r  ?= 0
          bne   sem_post       ; if r  != 0, retry
          dmb                  ; memory barrier
          bx    lr             ; return
```

Listing ([18, Example 1-6])

```
sem_wait: ldrex     r1, [ r0 ] ; s' = MEM[ &s ]
          cmp   r1, #0         ;     s' ?= 0
          beq   sem_wait       ; if s' == 0, retry
          sub   r1, r1, #1     ; s' = s' - 1
          strex r2, r1, [ r0 ] ; r <= MEM[ &s ] = s'
          cmp   r2, #0         ;     r  ?= 0
          bne   sem_wait       ; if r  != 0, retry
          dmb                  ; memory barrier
          bx    lr             ; return
```

Notes:

- Notice that this implementation performs a spin-lock, in the sense that while it blocks, it simply "spins" in a loop: there is, for example, no way to suspend the process if/when it blocks (which of course might be attractive). ARM [18, Example 1-6]) provide a slightly more complete example, where extra functionality can be included along these lines.
- The `dmb` instruction is a memory barrier: it *forces* the completion of pending memory accesses resulting from instructions before the barrier, and, as a result, a consistent (or up-to-date) view of the memory content.
  [18, Section 1.2.3], and [17, Figure A3.4.7] both explains the reason for including the `dmb` instruction in a code fragment as shown here. Consider `sem_wait`: since this function decrements the semaphore, it (eventually) acquires an associated resource. Intuitively, the `dmb` instruction is needed to make sure that acquisition process (e.g., storing the decremented semaphore) is complete before the resource is used or accessed.

- System V keyed semaphore API:
  - descriptor captured via type `int`,
  - related operations performed via
    - `semget` [16, Page 1836]:
      - allocate n-entry semaphore set if necessary

        $\texttt{flg} \ni \texttt{IPC\_CREAT} \Rightarrow$ allocate
        $\texttt{flg} \ni \texttt{IPC\_EXCL} \Rightarrow$ allocate or fail

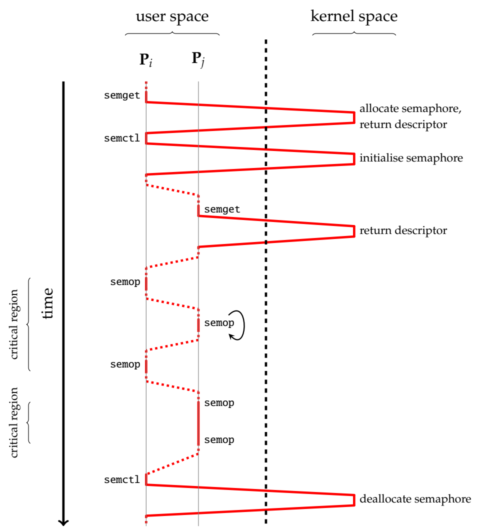
      - return descriptor.
    - `semctl` [16, Page 1833]:
      - control i-th semaphore in set, e.g.,

        $\texttt{cmd} = \texttt{IPC\_RMID} \Rightarrow$ deallocate
        $\texttt{cmd} = \texttt{IPC\_STAT} \Rightarrow$ get configuration
        $\texttt{cmd} = \texttt{SETVAL} \Rightarrow$ set value
        $\texttt{cmd} = \texttt{GETVAL} \Rightarrow$ get value

    - `semop` [16, Page 1839]:
      - update i-th semaphore in set, e.g.,

        $\texttt{op} < 0 \Rightarrow$ decrement
        $\texttt{op} = 0 \Rightarrow$ block
        $\texttt{op} > 0 \Rightarrow$ increment

      - block if necessary.



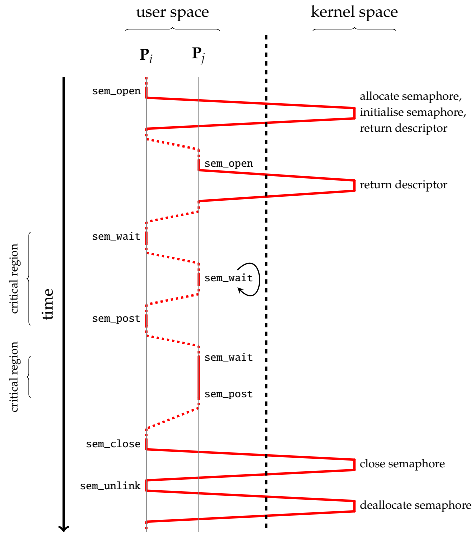
Notes:

IPC-related synchronisation: semaphore (8)
POSIX

- POSIX named semaphore API:
  - descriptor captured via type `sem_t`,
  - related operations performed via
    - `sem_open` [16, Page 1820]:
      - allocate semaphore if necessary

        `flg` $\ni$ `O_CREAT` $\Rightarrow$ allocate
        `flg` $\ni$ `O_EXCL` $\Rightarrow$ allocate or fail

      - initialise semaphore value if necessary,
      - return descriptor.
    - `sem_wait` [16, Page 1832]:
      - decrement semaphore value,
      - block if necessary.
    - `sem_post` [16, Page 1823]:
      - increment semaphore value.
    - `sem_close` [16, Page 1812]:
      - close semaphore (i.e., stop using it).
    - `sem_unlink` [16, Page 1830]:
      - unlink semaphore (i.e., deallocate it).



Notes:

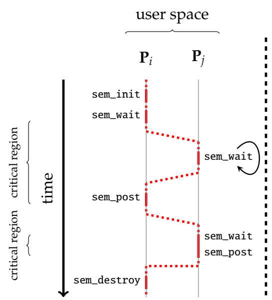
- The lack of interaction required between user and kernel space for named semaphores suggests this is an efficient option. It is, but there is a caveat: it requires *pre*-allocation of some shared memory for the semaphore. For synchronisation of threads doing so has no overhead, but for processes the kernel will need to be involved. However, the purpose of a named semaphore is often to synchronise access to a region of shared memory; the two mechanisms therefore mesh well, in the sense one can just allocate said region then use a small portion of it for the semaphore.
- It might not be obvious at face value, but notice the similarity between the function identifiers here and those within the file system API: both cases include `open`, `close` and `unlink`, for example, with largely similar semantics (iff. you consider a semaphore as a resource akin to a file, which of course for the named variant it more or less *is*).
- The API is somewhat more rich than shown here. For example, it includes additional entries such as:
  - `sem_getvalue` [16, Page 1816] offers a way to "peek" at the semaphore value, bypassing the normal interface (which does not expose the value itself).
  - `sem_timedwait` [16, Page 1825] is similar to `sem_wait`, but includes a timeout on the blocking behaviour.
  - `sem_trywait` [16, Page 1828] is similar to `sem_wait`, but is non-blocking: *if* it would block, it instead returns an error st. the caller can retry if need be.
- Some of the API is quite obtuse, in the sense it is unclear what exactly it does. `sem_destroy` is an clear example, whose behaviour is, on the most part, undefined or implementation defined: it obviously does make sense to offer a counterpart to `sem_init`, but, since the shared memory is pre-acquired, there is no need to deallocate it (e.g., in a similar way to `sem_unlink`).

IPC-related synchronisation: semaphore (8)
POSIX

- POSIX unnamed semaphore API:
  - descriptor captured via type `sem_t`,
  - related operations performed via
    - `sem_init` [16, Page 1818]:
      - initialise semaphore value

        `shared` $= 0 \Rightarrow$ shared between threads
        `shared` $\neq 0 \Rightarrow$ shared between processes

    - `sem_wait` [16, Page 1832]:
      - decrement semaphore value,
      - block if necessary.
    - `sem_post` [16, Page 1823]:
      - increment semaphore value.
    - `sem_destroy` [16, Page 1814]:
      - destroy semaphore (i.e., stop using it).



Notes:

- The lack of interaction required between user and kernel space for named semaphores suggests this is an efficient option. It is, but there is a caveat: it requires *pre*-allocation of some shared memory for the semaphore. For synchronisation of threads doing so has no overhead, but for processes the kernel will need to be involved. However, the purpose of a named semaphore is often to synchronise access to a region of shared memory; the two mechanisms therefore mesh well, in the sense one can just allocate said region then use a small portion of it for the semaphore.
- It might not be obvious at face value, but notice the similarity between the function identifiers here and those within the file system API: both cases include `open`, `close` and `unlink`, for example, with largely similar semantics (iff. you consider a semaphore as a resource akin to a file, which of course for the named variant it more or less *is*).
- The API is somewhat more rich than shown here. For example, it includes additional entries such as:
  - `sem_getvalue` [16, Page 1816] offers a way to "peek" at the semaphore value, bypassing the normal interface (which does not expose the value itself).
  - `sem_timedwait` [16, Page 1825] is similar to `sem_wait`, but includes a timeout on the blocking behaviour.
  - `sem_trywait` [16, Page 1828] is similar to `sem_wait`, but is non-blocking: *if* it would block, it instead returns an error st. the caller can retry if need be.
- Some of the API is quite obtuse, in the sense it is unclear what exactly it does. `sem_destroy` is an clear example, whose behaviour is, on the most part, undefined or implementation defined: it obviously does make sense to offer a counterpart to `sem_init`, but, since the shared memory is pre-acquired, there is no need to deallocate it (e.g., in a similar way to `sem_unlink`).

IPC-related synchronisation: semaphore (9)

Linux

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

```
struct semaphore {
        atomic_t count;
        int sleepers;
        wait_queue_head_t wait;
};
```

Notes:

IPC-related synchronisation: semaphore (10)

Linux

Listing (linux-2.6.10/include/asm-arm/locks.c)

```
#define __down_op(ptr,fail)                        \
        ({                                         \
        __asm__ __volatile__(                      \
        "@ down_op\n"                              \
"1:     ldrex   lr, [%0]\n"                        \
"       sub     lr, lr, %1\n"                      \
"       strex   ip, lr, [%0]\n"                    \
"       teq     ip, #0\n"                          \
"       bne     1b\n"                              \
"       teq     lr, #0\n"                          \
"       movmi   ip, %0\n"                          \
"       blmi    " #fail                            \
        :                                          \
        : "r" (ptr), "I" (1)                       \
        : "ip", "lr", "cc", "memory");             \
        })
```

Notes:

IPC-related synchronisation: semaphore (11)

Linux

Listing (`linux-2.6.10/include/asm-arm/locks.c`)

```
#define __up_op(ptr,wake)                        \
        ({                                       \
        __asm__ __volatile__(                    \
        "@ up_op\n"                              \
"1:     ldrex   lr, [%0]\n"                      \
"       add     lr, lr, %1\n"                    \
"       strex   ip, lr, [%0]\n"                  \
"       teq     ip, #0\n"                        \
"       bne     1b\n"                            \
"       teq     lr, #0\n"                        \
"       movle   ip, %0\n"                        \
"       blle    " #wake                          \
        :                                        \
        : "r" (ptr), "I" (1)                     \
        : "ip", "lr", "cc", "memory");           \
        })
```


Notes:

IPC-related notification: signal (1)

- Idea: signals relate to communication of *events*, e.g.,
  - process $\mathbf{P}_j$ registers a **signal handler** for a given signal type,
  - process $\mathbf{P}_i$ raises a signal of said type, via the kernel, with $\mathbf{P}_j$ as the target,
  - the kernel delivers the signal by invoking the handler.
- Example(s):
  1. $\mathbf{P}_i$ signals to $\mathbf{P}_j$ that it should terminate (e.g., due to Ctrl-C) via `SIGINT`.
  2. the kernel signals to $\mathbf{P}_j$ that it accessed an invalid address via `SIGBUS` or `SIGSEGV`.

Notes:

IPC-related notification: signal (1)

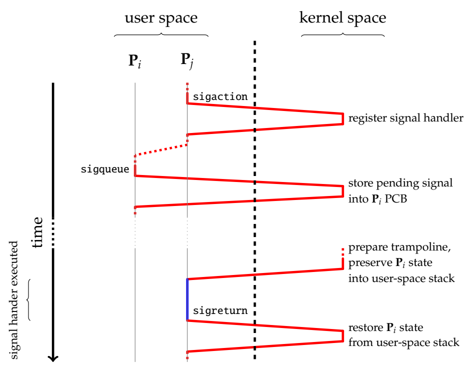
- Beware: signal delivery is *asynchronous*, meaning we need to
  - cater for reentrancy (e.g., if/when a signal is raised when executing a handler),
  - take care when interacting with the standard library (e.g., `write` can return `EINTR`),
  - take care with making updates to shared state

  noting the conceptual relationship with interrupt handling.

Notes:

IPC-related notification: signal (2)

- POSIX signal API:
  - handler captured via type
    `void (*h)( int id ),`
  - related operations performed via
    - `sigaction` [16, Page 1915]:
      - register handler.
    - `sigqueue` [16, Page 1944]:
      - enqueue signal, ready for delivery.
    - `sigreturn`:
      - return from signal handler.



Notes:

- There is a complication with this API, in the sense there is an existing and somewhat overlapping version specified by the C standard:
  - `signal` [16, Page 1937] is sort of the same as `sigaction`.
  - `kill` [16, Page 1199] is sort of the same as `sigqueue`, noting the identifier of the former is a misnomer: it can deliver *any* signal type, not just `SIGKILL`.
- The API is somewhat more rich than shown here. For example, it includes additional entries such as
  - `sigaltstack` [16, Page 1924] supports use of an alternate stack (i.e., alternate to the default, user space stack) wrt. signal delivery.

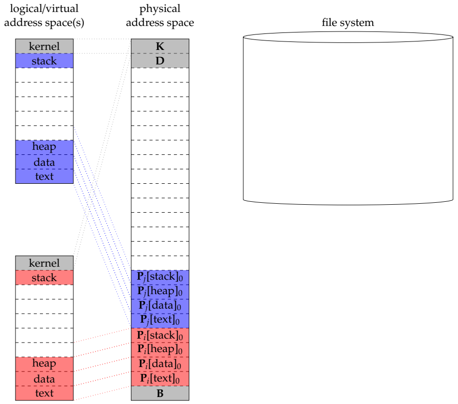
  and supports (at least) two mechanisms to control signal delivery:

  1. `sigaction` allows the specification of a signal mask that blocks a set of signals (i.e., prevents their delivery) while the given signal is being handled; doing so basically avoids various issues of reentrancy. The signal mask can later be updated using, for example, `sigaddset` [16, Page 1923] and `sigdelset` [16, Page 1926] to add and remove a signal from the signal mask.
  2. `sigaction` allows the specification of various flags (or options). An important example is `SA_RESTART`: this can be used to specify certain primitives (e.g., `write`) should be restarted if interrupted by a signal (vs. failing via the `EINTR` error).
- There are two special-purpose signal handlers, namely `SIG_DFL` which registers the default handler for a given type (e.g., something to terminate the process on delivery of `SIGINT`, and `SIG_IGN` to which registers a null or empty handler for a given type (i.e., ignores signals of that type).
- Some signals cannot be handled within this mechanism: `SIGKILL` and `SIGSTOP` are two examples. For these signals specifically, the argument is they should not be handled by the process because they relate to process management by the kernel. For example, there must be a way for the kernel to terminate unresponsive processes; if the process could handle and then *ignore* `SIGKILL`, this would become impossible in any reliable sense.
- The diagram hints at a few subtle details:
  - First, where is the execution context for $\mathbf{P}_i$ preserved when it is suspended? One obvious answer is to use the PCB for $\mathbf{P}_i$, as would be the case if suspending the process in order to then resume another (i.e., a context switch). However, this could only work iff. we guarantee there is never a context switch *during* execution of the handler: if there were the state would be overwritten by that of the handler, offering no way to resume the process where it was originally suspended. As a result, the execution context for $\mathbf{P}_i$ is preserved by the kernel on the associated user space stack.
  - Next, how do we ensure execution of $\mathbf{P}_i$ resumes where it was suspended? The restoration of the preserved execution context is performed via the `sigreturn` system call (noting this is *not* part of the API: most kernels support a similar system call to facilitate signal delivery, but it is not standardised). We therefore have two main options, namely

    1. assume the handler invokes `sigreturn`, or
    2. force invocation of `sigreturn` from a wrapper function around the handler called a **trampoline**, the instructions for which can *also* be constructed and stored dynamically on the user space stack.

IPC-related communication: shared memory/mapping (1)

- Idea:



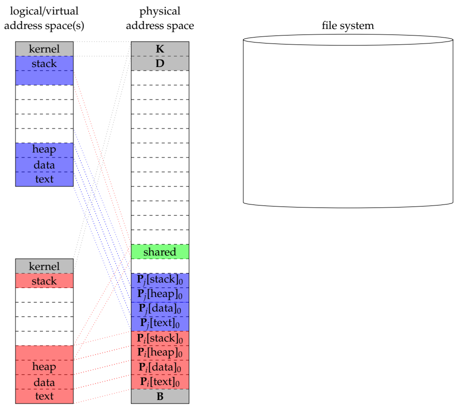
st.

- − synchronisation needs to be explicit
- ± communication is unstructured (i.e., byte-oriented)
- \+ communication is (relatively) efficient
- \+ communication is bi-directional
- \+ supports n-to-m communication

Notes:

IPC-related communication: shared memory/mapping (1)

- Idea:



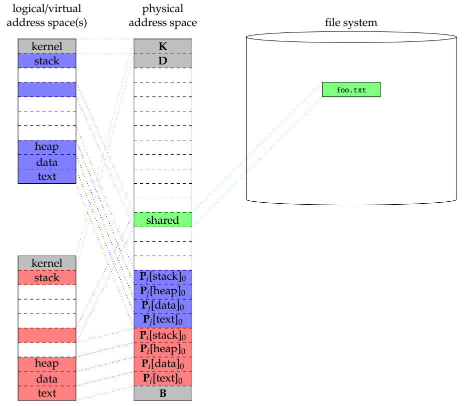
st.

- − synchronisation needs to be explicit
- ± communication is unstructured (i.e., byte-oriented)
- \+ communication is (relatively) efficient
- \+ communication is bi-directional
- \+ supports n-to-m communication

Notes:

IPC-related communication: shared memory/mapping (1)

- **Idea**:



  st.

  – synchronisation needs to be explicit
  ± communication is unstructured (i.e., byte-oriented)
  + communication is (relatively) efficient
  + communication is bi-directional
  + supports n-to-m communication

Notes:

IPC-related communication: shared memory/mapping (2)
System V

- System V keyed shared memory API:
  - descriptor captured via type `int`,
  - related operations performed via
    - `shmget` [16, Page 1911]:
      - allocate n-byte shared memory region if necessary

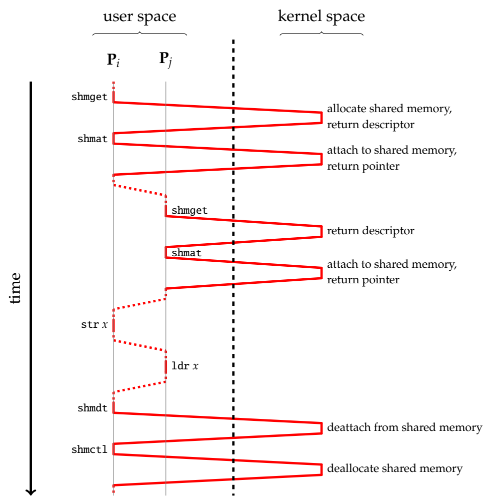
        `flg` ∋ `IPC_CREAT` ⇒ allocate
        `flg` ∋ `IPC_EXCL` ⇒ allocate or fail

      - return descriptor.
    - `shmat` [16, Page 1905]:
      - attach to shared memory region

        `flg` ∋ `SHM_EXEC` ⇒ execute permission
        `flg` ∋ `SHM_RDONLY` ⇒ read permission

      - return pointer.
    - `shmdt` [16, Page 1909]:
      - detach from shared memory region.
    - `shmctl` [16, Page 1907]:
      - control shared memory region

        `cmd` = `IPC_RMID` ⇒ deallocate
        `cmd` = `IPC_STAT` ⇒ get configuration



Notes:

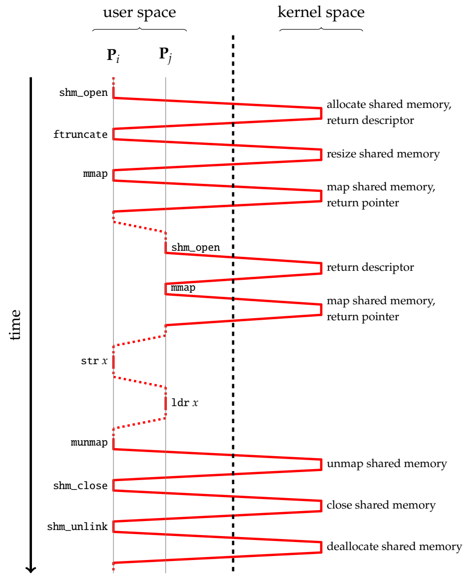
IPC-related communication: shared memory/mapping (3)

POSIX

- POSIX named shared memory API:
  - descriptor captured via type `int`,
  - related operations performed via
    - `shm_open` [16, Page 1898]:
      - allocate n-byte shared memory region if necessary

        `flg` $\ni$ `O_CREAT` $\Rightarrow$ allocate
        `flg` $\ni$ `O_EXCL` $\Rightarrow$ allocate or fail
        `flg` $\ni$ `O_RDWR` $\Rightarrow$ read/write permission
        `flg` $\ni$ `O_RDONLY` $\Rightarrow$ read permission

      - return descriptor.
    - `mmap` [16, Page 1309]:
      - map shared memory region into virtual address space.
      - return pointer.
    - `munmap` [16, Page 1357]:
      - unmap shared memory region from virtual address space.
    - `shm_close`:
      - close shared memory region (i.e., stop using it).
    - `shm_unlink` [16, Page 1903]:
      - unlink shared memory region (i.e., deallocate it).



Notes:

IPC-related communication: shared memory/mapping (4)

Linux

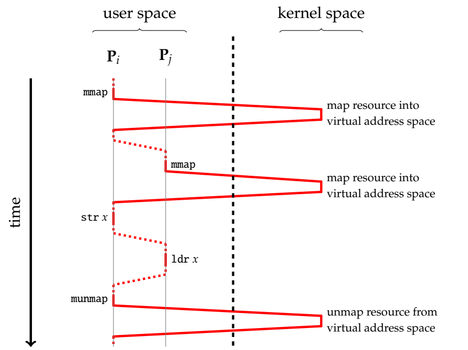
- Linux shared mapping API:
  - `mmap` [16, Page 1309]:
    - map resource into virtual address space

      `flg` $\ni$ `MAP_SHARED` $\Rightarrow$ shared mapping
      `flg` $\ni$ `MAP_PRIVATE` $\Rightarrow$ unshared mapping
      `flg` $\ni$ `MAP_ANONYMOUS` $\Rightarrow$ memory mapping

    - initialise access permissions

      `prot` $\ni$ `PROT_EXEC` $\Rightarrow$ execute permission
      `prot` $\ni$ `PROT_READ` $\Rightarrow$ read permission
      `prot` $\ni$ `PROT_WRITE` $\Rightarrow$ write permission

    - return pointer.
  - `munmap` [16, Page 1357]:
    - unmap resource from virtual address space.

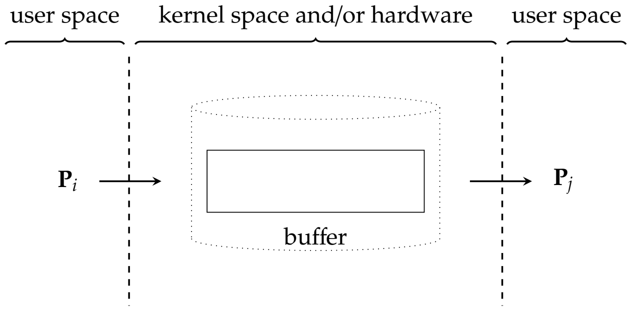


Notes:

- `mmap` is single, yet very versatile, interface: one point to keep in mind is that, intuitively, it offers a way for the programmer to control the virtual memory mechanism (to some extent).
- The API is somewhat more rich than shown here. For example, it includes additional entries such as:
  - `msync` [16, Page 1352] forces any pending (e.g., buffered) stores associated with a mapped region to complete, thus writing data to the underlying file; this is conceptually the same as `fflush`, which does a similar sort of thing for buffered data wrt. I/O.
  - `mprotect` [16, Page 1319] alters the access permissions for a mapped region, e.g., adding or removing permissions to read, write or execute it.
  - `mlock` [16, Page 1305] and `munlock` [16, Page 1355] respectively lock or unlock a memory region; put simply, this means the (physical) pages frames that back the associated (virtual) pages are prevented from being swapped-out (i.e., they remain in physical memory).
- By default, `mmap` creates a file mapping: this maps the contents of a file into the virtual address space; the `MAP_ANONYMOUS` makes use of memory instead, st. there is no (persistent) file backing the mapping. A given mapping can be private or shared, where interpreting the latter needs some care. More specifically, for a file mapping it implies two unrelated processes can create a shared memory region by mapping the same file; for a memory mapping this makes no sense of course, *but* the flag does imply that the mapping persists across use of `fork` so can be used to form a shared memory region between *related* processes.

IPC-related communication: pipe (1)

- **Idea**:



st.

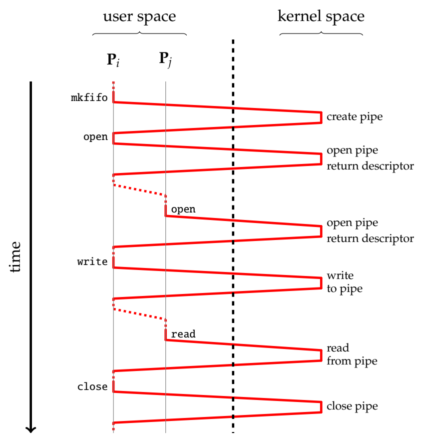
+ synchronisation is implicit
± communication is unstructured (i.e., byte-oriented)
− communication is (relatively) inefficient (i.e., vs. shared memory)
− communication is uni-directional
− supports 1-to-1 communication

Notes:

- Read this diagram as attempting to describe a "buffer that looks like a file".

IPC-related communication: pipe (2)
POSIX

- POSIX named pipe API:
  - descriptor captured via type `int`,
  - related operations performed via
    - `mkfifo` [16, Page 1295]:
      - allocate pipe,
      - initialise access permissions.
    - `open` [16, Page 1379]:
      - open pipe,
      - return descriptor.
    - `write` [16, Page 2263]:
      - write data to pipe,
      - block if necessary.
    - `read` [16, Page 1737]:
      - read data from pipe,
      - block if necessary.
    - `close` [16, Page 676]:
      - close pipe (i.e., stop using it).
    - `unlink` [16, Page 2154]:
      - unlink pipe (i.e., deallocate it).

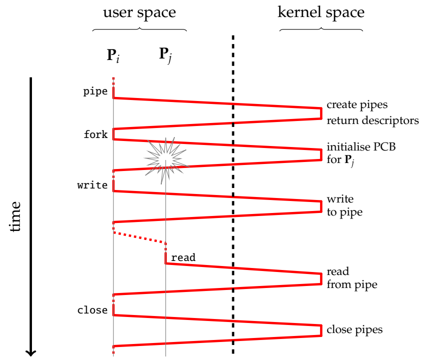


Notes:

- Note that a named pipe is more commonly termed a FIFO, which describes the semantics and explains the function identifier `mkfifo`.
- On Linux at least, `pipe2` is an alternative to `pipe` that allows specification of various flags.

IPC-related communication: pipe (2)
POSIX

- POSIX unnamed pipe API:
  - descriptor captured via type `int`,
  - related operations performed via
    - `pipe` [16, Page 1400]:
      - allocate pipes,
      - return descriptors.
    - `write` [16, Page 2263]:
      - write data to pipe,
      - block if necessary.
    - `read` [16, Page 1737]:
      - read data from pipe,
      - block if necessary.
    - `close` [16, Page 676]:
      - close pipe (i.e., stop using it).

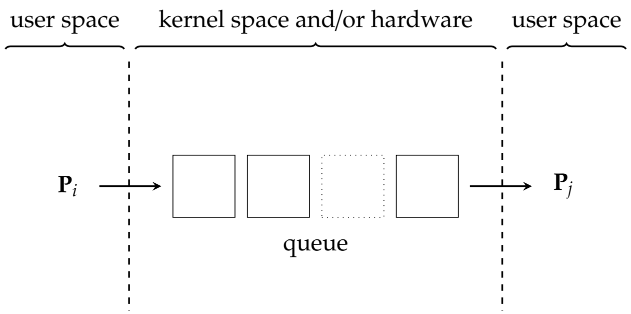


Notes:

- Note that a named pipe is more commonly termed a FIFO, which describes the semantics and explains the function identifier `mkfifo`.
- On Linux at least, `pipe2` is an alternative to `pipe` that allows specification of various flags.

IPC-related communication: message queue (1)

- Idea:



st.

\+ synchronisation is implicit
± communication is structured (i.e., datagram-oriented)
− communication is (relatively) inefficient (i.e., vs. shared memory)
− communication is bi-directional
− supports n-to-m communication

Notes:

IPC-related communication: message queue (2)

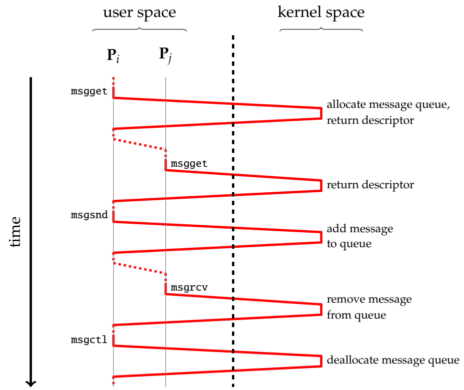
System V

- System V keyed message queue API:
  - descriptor captured via type `int`,
  - related operations performed via
    - `msgget` [16, Page 1344]:
      - allocate message queue if necessary

        `flg` $\ni$ `IPC_CREAT` $\Rightarrow$ allocate
        `flg` $\ni$ `IPC_EXCL` $\Rightarrow$ allocate or fail

      - return descriptor.
    - `msgsnd` [16, Page 1349]:
      - add data to message queue,
      - block if necessary.
    - `msgrcv` [16, Page 1346]:
      - remove data from message queue,
      - block if necessary.
    - `msgctl` [16, Page 1342]:
      - control message queue

        `cmd` = `IPC_RMID` $\Rightarrow$ deallocate
        `cmd` = `IPC_STAT` $\Rightarrow$ get configuration



Notes:

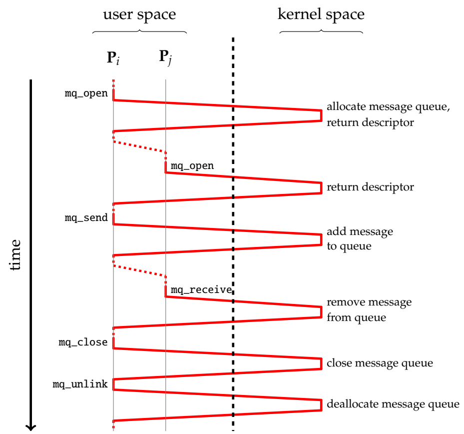
IPC-related communication: message queue (3)

POSIX

- POSIX named message queue API:
  - descriptor captured via type `mqd_t`,
  - related operations performed via
    - `mq_open` [16, Page 1327]:
      - allocate message queue if necessary

        `flg` $\ni$ `O_CREAT` $\Rightarrow$ allocate
        `flg` $\ni$ `O_EXCL` $\Rightarrow$ allocate or fail
        `flg` $\ni$ `O_RDWR` $\Rightarrow$ read/write permission
        `flg` $\ni$ `O_RDONLY` $\Rightarrow$ read permission
        `flg` $\ni$ `O_WRONLY` $\Rightarrow$ write permission
        `flg` $\ni$ `O_NOBLOCK` $\Rightarrow$ non-blocking mode

      - return descriptor.
    - `mq_send` [16, Page 1333]:
      - add data to message queue,
      - block if necessary.
    - `mq_receive` [16, Page 1330]:
      - remove data from message queue,
      - block if necessary.
    - `mq_close` [16, Page 1321]:
      - close message queue (i.e., stop using it).
    - `mq_unlink` [16, Page 1339]:
      - unlink message queue (i.e., deallocate it).

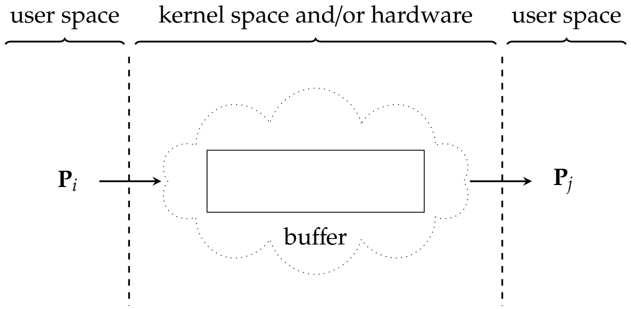


Notes:

- The API is somewhat more rich than shown here. For example, it includes additional entries such as:
  - `mq_getattr` [16, Page 1322] and `mq_setattr` [16, Page 1335] allow access to attributes associated with a given message queue, including, for example, the maximum number of messages it is able to accommodate.
  - `mq_notify` [16, Page 1324] allows a process to register a callback function, akin to a signal handler, which is then invoked when a message is added to a given message queue: one can imagine server-like functionality based on this mechanism, whereby the server waits until a message is made available then processes it somehow.

IPC-related communication: domain socket (1)

- Idea:



st.

\+ synchronisation is implicit
± communication is unstructured (i.e., byte-oriented)
− communication is (relatively) inefficient (i.e., vs. shared memory)
\+ communication is bi-directional
− supports 1-to-1 communication

Notes:

- Read this diagram as attempting to describe a "buffer that looks like a network interface". This is important, because it is *tempting* to consider a domain socket as connecting to some form of "local" or *intra*-host network. Indeed, this is what the diagram might suggest. However attractive doing so is wrt. providing an analogy, it remains vital to remember that it is misleading wrt. implementation of the concept.
  Although, from a user-facing side, the API is like the normal network socket API, the kernel-facing side can be significantly more simple by virtue of the fact all communication is inter-process. This means, for example, a domain socket is fundamentally different from network communication using a "local" network resulting from a loop-back interface; the term domain is, likewise, unrelated to the networking concept (e.g., a domain name).

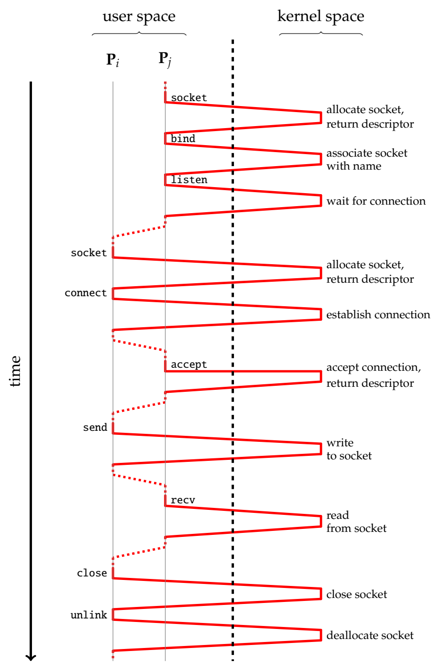
IPC-related communication: domain socket (2)

POSIX

- POSIX named domain socket API:
  - descriptor captured via type `int`,
  - related operations performed via
    - `socket` [16, Page 1968]:
      - allocate socket

`domain` = `AF_UNIX`	⇒	IPC
`domain` = `AF_INET`	⇒	IPv4 network
`domain` = `AF_INET6`	⇒	IPv4 network
`type` = `SOCK_STREAM`	⇒	stream model
`type` = `SOCK_DGRAM`	⇒	datagram model

      - return descriptor.
    - `send` [16, Page 1844] (or `write` [16, Page 2263]):
      - write data to socket,
      - block if necessary.
    - `recv` [16, Page 1759] (or `read` [16, Page 1737]):
      - read data from socket,
      - block if necessary.
    - `close` [16, Page 676]:
      - close socket (i.e., stop using it).
    - `unlink` [16, Page 2154]:
      - unlink socket (i.e., deallocate it).



Notes:

- Some of the API is not described here, due to the restriction on space. In particular, the following are excluded
  - `bind` [16, Page 616] associates a socket with an address,
  - `listen` [16, Page 1225] instructs the kernel for listen for incoming connections on a given socket,
  - `connect` [16, Page 690] instructs the kernel to form an outgoing connection on a given socket,
  - `accept` [16, Page 559] instructs the kernel to accept an incoming connection on a given socket,

  which collectively act to establish a connection, over the domain socket, between $\mathbf{P}_i$ (acting as a client) and $\mathbf{P}_j$ (acting as a server).

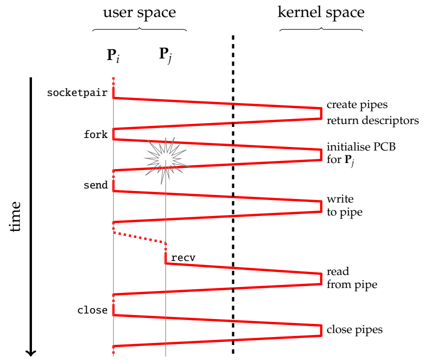
IPC-related communication: domain socket (2)

POSIX

- POSIX unnamed domain socket API:
  - descriptor captured via type `int`,
  - related operations performed via
    - `socketpair` [16, Page 1970]:
      - allocate sockets

        `domain = AF_UNIX` ⇒ IPC
        `domain = AF_INET` ⇒ IPv4 network
        `domain = AF_INET6` ⇒ IPv4 network
        `type = SOCK_STREAM` ⇒ stream model
        `type = SOCK_DGRAM` ⇒ datagram model

      - return descriptors.
    - `send` [16, Page 1844] (or `write` [16, Page 2263]):
      - write data to socket,
      - block if necessary.
    - `recv` [16, Page 1759] (or `read` [16, Page 1737]):
      - read data from socket,
      - block if necessary.



Notes:

- Some of the API is not described here, due to the restriction on space. In particular, the following are excluded
  - `bind` [16, Page 616] associates a socket with an address,
  - `listen` [16, Page 1225] instructs the kernel for listen for incoming connections on a given socket,
  - `connect` [16, Page 690] instructs the kernel to form an outgoing connection on a given socket,
  - `accept` [16, Page 559] instructs the kernel to accept an incoming connection on a given socket,

  which collectively act to establish a connection, over the domain socket, between $\mathbf{P}_i$ (acting as a client) and $\mathbf{P}_j$ (acting as a server).

Conclusions

- Take away points:
  - IPC is important, representing a core service delivered by kernel ...
  - ... *good* IPC is tricky, wrt.
  1. interface
     - large design space,
     - (ideally) needs to be flexible, uniform, etc.
     - can unify other abstractions (e.g., files, sockets),
     - should promote correctness,
  2. implementation
     - large design space,
     - needs to be efficient: pure overhead wrt. concurrent computation,
     - can share other mechanisms (e.g., files, sockets),
     - should enforce correctness,

  meaning *multiple*, complementary variants are the norm.

Notes:

- A non-exhaustive list of topics *not* covered, or covered in a more superficial level than ideal, include
  - extension of IPC concepts to **Remote Procedure Call (RPC)**,
  - higher-level, user- and application-oriented proxies, such as D-Bus [1].

Conclusions

- Take away points:
  - A study of existing standards highlights design philosophy, e.g.,

$$\text{System V} \mapsto \begin{cases} \texttt{Xget} & \Rightarrow \text{allocate resource and descriptor} \\ \texttt{Xctl} & \Rightarrow \text{configure resource, plus deallocate descriptor and resource} \\ \texttt{Xop} & \Rightarrow \text{general-purpose operation} \end{cases}$$

$$\text{POSIX} \mapsto \begin{cases} \texttt{X\_open} & \Rightarrow \text{allocate resource and descriptor, plus configure resource} \\ \texttt{X\_Y} & \Rightarrow \text{special-purpose operation} \\ \texttt{X\_close} & \Rightarrow \text{deallocate descriptor} \\ \texttt{X\_unlink} & \Rightarrow \text{deallocate resource} \end{cases}$$

Notes:

- A non-exhaustive list of topics *not* covered, or covered in a more superficial level than ideal, include
  - extension of IPC concepts to **Remote Procedure Call (RPC)**,
  - higher-level, user- and application-oriented proxies, such as D-Bus [1].

Additional Reading

- A.B. Downey. *The Little Book of Semaphores*. `http://greenteapress.com/wp/semaphores`.
- M. Kerrisk. "Chapter 47: System V semaphores". In: *The Linux Programming Interface*. 6th ed. No Starch Press, 2010.
- M. Kerrisk. "Chapter 53: POSIX semaphores". In: *The Linux Programming Interface*. 6th ed. No Starch Press, 2010.
- M. Kerrisk. "Chapter 20: Signals: fundamental concepts". In: *The Linux Programming Interface*. 6th ed. No Starch Press, 2010.
- M. Kerrisk. "Chapter 21: Signals: signal handlers". In: *The Linux Programming Interface*. 6th ed. No Starch Press, 2010.
- M. Kerrisk. "Chapter 49: Memory mappings". In: *The Linux Programming Interface*. 6th ed. No Starch Press, 2010.
- M. Kerrisk. "Chapter 48: System V shared memory". In: *The Linux Programming Interface*. 6th ed. No Starch Press, 2010.
- M. Kerrisk. "Chapter 54: POSIX shared memory". In: *The Linux Programming Interface*. 6th ed. No Starch Press, 2010.
- M. Kerrisk. "Chapter 46: System V message queues". In: *The Linux Programming Interface*. 6th ed. No Starch Press, 2010.
- M. Kerrisk. "Chapter 52: POSIX message queues". In: *The Linux Programming Interface*. 6th ed. No Starch Press, 2010.
- M. Kerrisk. "Chapter 44: Pipes and FIFOs". In: *The Linux Programming Interface*. 6th ed. No Starch Press, 2010.
- M. Kerrisk. "Chapter 57: Sockets: UNIX domain". In: *The Linux Programming Interface*. 6th ed. No Starch Press, 2010.
- R. Love. "Chapter 4: Advanced file I/O". . In: *Linux System Programming*. 2nd ed. O'Reilly, 2013.
- R. Love. "Chapter 10: Signals". In: *Linux System Programming*. 2nd ed. O'Reilly, 2013.

Notes:

References

[1] *Wikipedia: D-Bus*. URL: http://en.wikipedia.org/wiki/D-Bus (see pp. 76, 78).

[2] A.B. Downey. *The Little Book of Semaphores*. http://greenteapress.com/wp/semaphores (see p. 79).

[3] M. Kerrisk. "Chapter 20: Signals: fundamental concepts". In: *The Linux Programming Interface*. 6th ed. No Starch Press, 2010 (see p. 79).

[4] M. Kerrisk. "Chapter 21: Signals: signal handlers". In: *The Linux Programming Interface*. 6th ed. No Starch Press, 2010 (see p. 79).

[5] M. Kerrisk. "Chapter 44: Pipes and FIFOs". In: *The Linux Programming Interface*. 6th ed. No Starch Press, 2010 (see p. 79).

[6] M. Kerrisk. "Chapter 46: System V message queues". In: *The Linux Programming Interface*. 6th ed. No Starch Press, 2010 (see p. 79).

[7] M. Kerrisk. "Chapter 47: System V semaphores". In: *The Linux Programming Interface*. 6th ed. No Starch Press, 2010 (see p. 79).

[8] M. Kerrisk. "Chapter 48: System V shared memory". In: *The Linux Programming Interface*. 6th ed. No Starch Press, 2010 (see p. 79).

[9] M. Kerrisk. "Chapter 49: Memory mappings". In: *The Linux Programming Interface*. 6th ed. No Starch Press, 2010 (see p. 79).

[10] M. Kerrisk. "Chapter 52: POSIX message queues". In: *The Linux Programming Interface*. 6th ed. No Starch Press, 2010 (see p. 79).

[11] M. Kerrisk. "Chapter 53: POSIX semaphores". In: *The Linux Programming Interface*. 6th ed. No Starch Press, 2010 (see p. 79).

[12] M. Kerrisk. "Chapter 54: POSIX shared memory". In: *The Linux Programming Interface*. 6th ed. No Starch Press, 2010 (see p. 79).

[13] M. Kerrisk. "Chapter 57: Sockets: UNIX domain". In: *The Linux Programming Interface*. 6th ed. No Starch Press, 2010 (see p. 79).

[14] R. Love. "Chapter 10: Signals". In: *Linux System Programming*. 2nd ed. O'Reilly, 2013 (see p. 79).

[15] R. Love. "Chapter 4: Advanced file I/O". In: *Linux System Programming*. 2nd ed. O'Reilly, 2013 (see p. 79).

Notes:

References

[16] *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 pp. 27, 29–32, 43, 44, 51, 53, 55, 56, 59, 61, 65, 67, 68, 71–74).

[17] ARM Limited. *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition*. Tech. rep. DDI-0406C. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406c/index.html. 2014 (see pp. 16, 17, 19, 22, 24, 26).

[18] ARM Limited. *ARM Synchronization Primitives*. Tech. rep. DHT-0008A. http://infocenter.arm.com/help/topic/com.arm.doc.dht0008a/index.html. 2009 (see pp. 13, 17, 19, 25, 26).

[19] E.W. Dijkstra. *Over de sequentialiteit van procesbeschrijvingen*. Tech. rep. EWD-35. http://www.cs.utexas.edu/users/EWD. 1962 (approx.) (See pp. 9–12).

[20] E.H. Jensen, G.W. Hagensen, and J.M. Broughton. *A new approach to exclusive data access in shared memory multi-processors*. Tech. rep. UCRL-97663. Lawrence Livermore National Laboratory, 1987 (see pp. 10, 12).

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

Notes: