Concurrent Computing (Operating Systems)

Daniel Page

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

February 8, 2016

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:

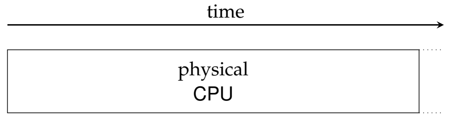
Notes:

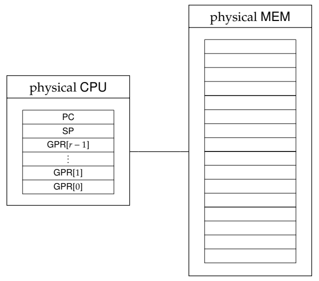
Concept: *virtualise* the processor

- 1 process *does* have dedicated access to the physical processor.
- We know execution is st.

```
 ┌→ fetch         fetch    fetch    fetch    ···
 │  decode   ≡    decode   decode   decode   ···
 └─ execute       execute  execute  execute  ···
```

i.e.,





Notes:

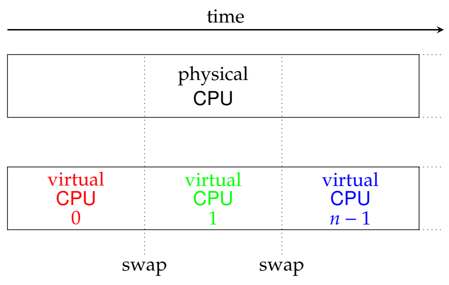
- By saying the physical processor swaps between processes, we basically mean it ensures the right **execution context** at the right time: if, for example, the physical processor uses the PC for some *i*-th process, the next instruction fetched and then executed will therefore be associated with *that* process rather than some other. This is the essence of **multi-tasking**.
- Since we use the POSIX standard as an example throughout, you might want to refer to the specific (and extensive) associated definitions *it* uses: pertinent examples include
  - program [1, Section 3.300],
  - process [1, Section 3.289],
  - thread [1, Section 3.396], and
  - signal [1, Section 3.344].

Concept: *virtualise* the processor

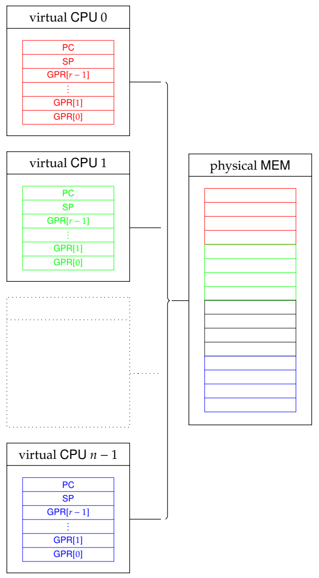
- *n* processes *cannot* have dedicated access to the physical processor ...
- ... but *if* execution could be st.

```
 ┌→ fetch         fetch    fetch    fetch    ···
 │  decode   ≡    decode   decode   decode   ···
 └─ execute       execute  execute  execute  ···
```

i.e.,



then they'd *appear* to.



Notes:

- By saying the physical processor swaps between processes, we basically mean it ensures the right **execution context** at the right time: if, for example, the physical processor uses the PC for some *i*-th process, the next instruction fetched and then executed will therefore be associated with *that* process rather than some other. This is the essence of **multi-tasking**.
- Since we use the POSIX standard as an example throughout, you might want to refer to the specific (and extensive) associated definitions *it* uses: pertinent examples include
  - program [1, Section 3.300],
  - process [1, Section 3.289],
  - thread [1, Section 3.396], and
  - signal [1, Section 3.344].

Concepts (1)

Definition (uni-/multi-{programming,processing,tasking})

The terms **uni-programming** and **multi-programming** are used, respectively, to describe cases where one or many programs execute simultaneously. In the latter case, execution could be

- parallel (or *truly*-parallel), e.g., as realised via **multi-processing**, or
- concurrent (or *pseudo*-parallel), e.g., as realised via **multi-tasking**.

Notes:

- The terminology here can be confusing, and is made less exact in certain contexts. The basic idea is as follows:
  - Under a multi-programming kernel, one or more programs are resident in memory (at the same time); at a given point in time, any one can be executed. This contrasts with uni-programming, where only one program can ever be resident and hence executed.
  - The similar sounding terms uni- and multi-*processing* relate to hardware rather than software: a multi-processing system will have many physical processors.
  - As such, multi-processing is one way to realise multi-programming. Another way is multi-*tasking*, where many programs share one given processor: this is essentially what **time-sharing** or **time-slicing** means.

Concepts (2)

Definition (process)

A **process** is an active instance of a given, passive program image. Each process constitutes

1. $n \geq 1$ **execution contexts** (viz. **threads**), each for an independent instruction stream, *plus*
2. associated state, i.e.,
   - an address space, and
   - a set of resources

   which is shared between those threads.

Definition (context switch)

A **context switch** is the act of, or mechanism for, changing the active execution context: performing a context switch will typically involve

- suspending execution of one process, then
- resuming execution of another process.

Notes:

- A more intuitive way to separate the concepts of processes and threads, is via their role: the former is really an entity used to group related resources that support execution, whereas the latter is an entity that captures an independent instruction stream *being* (or at least having the potential to be) executed by the processor.
- The terminology to describe processes can change a little depending on the context. For example, in batch processing systems it is common to hear the term **job** instead (also occurring in various UNIX-related contexts, e.g., the BASH job control commands). Also note that Linux uses the term **task** specifically st. traditional meaning implied by the terms process and thread can be avoided.
  In addition, keep in mind that processes and threads are may be termed heavy-weight and light-weight processes respectively: the reason to do so, and hence the meaning of the terms, depends on the context to some extent. One interpretation is that a light-weight process is a thread supported in user space. Another is simply that because threads share the resources of an associated process, their representation includes less (than said process) and so is lighter-weight; they are typically easier to create for the same reason (given no need to allocate said resources).
- If you want, you could indulge in some program vs. process philosophy:
  - A process is "*more than*" a program, since the former is a (stateful) instance of the latter: there can be more than one process all stemming from the same program (e.g., more than one instance of `emacs`).
  - A program is "*more than*" a process, in the sense the former captures all things that could happen, but the latter captures a specific case of what *is* happening; it is sometimes true that one program forms multiple processes (e.g., executing `gcc` might execute `cpp`, `cc1` and so on).

POSIX(ish) Realisation (1) – Representation

- Each process is represented by the kernel
  - in a **process table**,
  - each entry in which is a data structure termed a **Process Control Block (PCB)**

e.g.,

Process management	Memory management	Resource management
processor state	MMU state	user ID
process ID	text segment info.	group ID
process status	data segment info.	working directory
process hierarchy	stack segment info.	file descriptors
scheduling info.		
signalling info.		
accounting info.		
⋮	⋮	⋮

noting the entries are

- *very* kernel- and hardware-specific (so these are *examples* only), and
- divided into per-process and per-thread.

Notes:

- A process, user and group ID are almost always acronym'ised as PID, UID and GID; depending on how they are implemented, thread-specific versions might also exist, e.g., TID for thread ID.
- It should be clear that for a thread to be deemed independent, there must be a) an independent PC (and register file state more generally), plus b) an independent SP and stack region. This fact is reflected in how the entries are split between per-process and per-thread.
  The latter requirement may not be obvious, but exists for essentially the same reason we maintain a stack frame for nested function call instances: the frame, or **activation record**, captures the state of that instance in the same way. That is, if a caller instance calls some other callee function, the latter is allocated a new frame; when the callee returns and unwinds the callee frame, the old caller frame is intact, meaning the caller instance can continue executing. The same is true of threads, in the sense that if execution of one thread is suspended and another resumed, we want the latter thread to resume in the same state as when it was last suspended, *not* corrupted by whatever the last executing thread was.

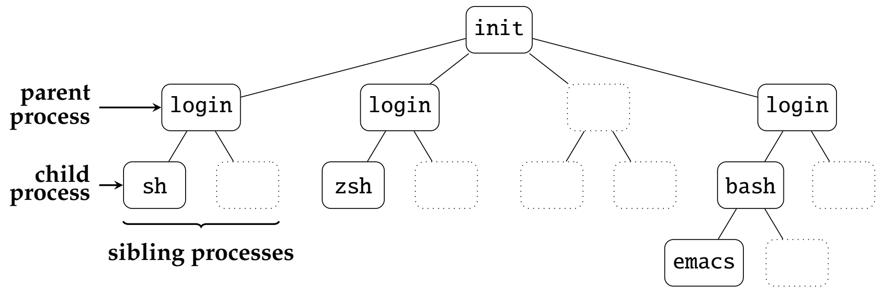
POSIX(ish) Realisation (2) – Representation

- POSIX says processes are organised
  1. into a **process hierarchy** [1, Sections 3.93 and 3.264], namely a tree, *and*
  2. **process groups** [1, Section 3.290] can be formed, e.g., to support collective communication.

Notes:

POSIX(ish) Realisation (2) – Representation

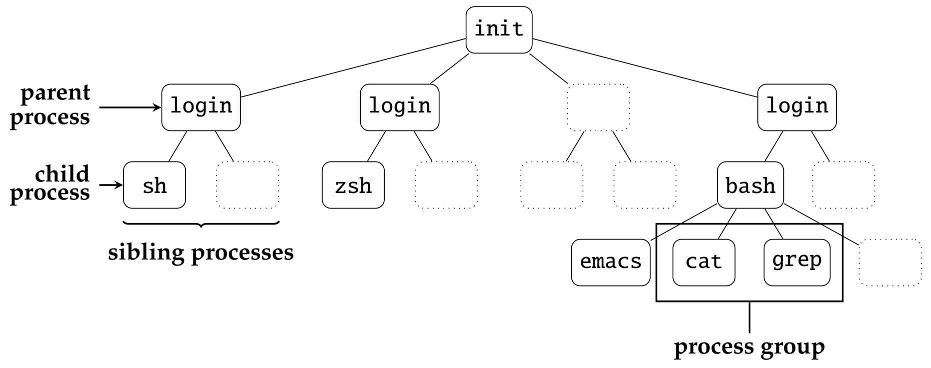
- Example:
  - we have three logged-in users,
  - one of whom is executing an instance of `emacs`.



Notes:

POSIX(ish) Realisation (2) – Representation

- Example:
  - we have three logged-in users,
  - one of whom is executing an instance of `emacs`,
  - and *then* executes `cat foo.txt | grep bar`



Notes:

POSIX(ish) Realisation (3) – Representation

Definition (orphaned process)

An **orphaned process** is one which is still executing even though it has no parent (i.e., the parent has terminated).

Definition (zombie process)

A **zombie process** (or **defunct process**) is one whose parent has (been) terminated, but still has an active PCB.

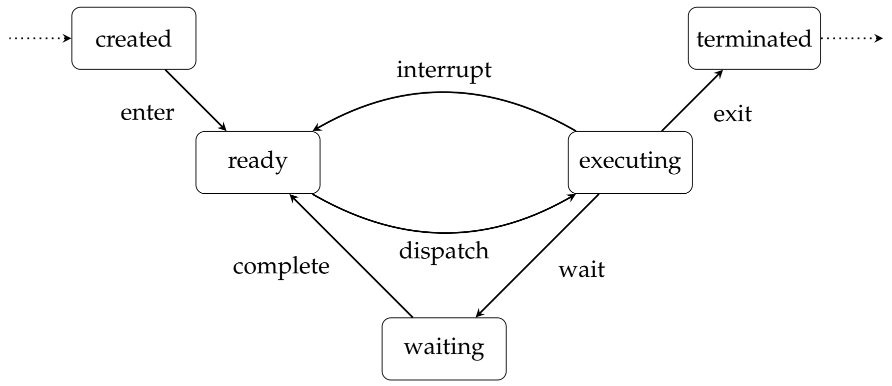
Definition (daemon process)

A **foreground process** (resp. **background process**) executes with (resp. without) direct interaction with a user. Uses for a background process (which may be termed a **daemon process**) include logging or maintenance services.

Notes:

POSIX(ish) Realisation (4) – Representation

- As execution progresses, the **process status** changes



under control of a (or the) **scheduler**:

- one or more **scheduling queues** keeps track of processes in each state: for example
- a **ready queue** captures processes ready to be executed, and
- a per-device **device queue** captures processes waiting for I/O operations to complete.

Notes:

- There is no definitive set of correct terms for nodes and edges in an FSM-like diagram of this sort: the FSM itself, and said terms depend on the kernel in question and level of detail. For instance, you can see a simpler FSM in [7, Figure 2-2] and different terms in [5, Figure 3.2]. The rationale for those shown is as follows:
  - The terms ready and executing are often replaced by runnable and running; given the use of execute vs. run elsewhere, the terms used here are mainly for consistency, with ready indicating readiness for execution vs. *being* executed.
  - The terms dispatch and interrupt might be confusing. The former is due to the fact the short-term scheduler uses the so-called the dispatcher to pass control to the process it has selected, i.e., perform the actual context switch; the latter captures the fact execution is interrupted, which *is* often because of an interrupt occurring, but also conflates the two unnecessarily to some extent.
    Either way, it is important to see that dispatch vs. schedule is a separation of mechanism from policy: dispatching is doing the context switch to have some process executed next, while scheduling is deciding which process that is.

  Likewise, other descriptions may use process state to refer to what is here termed process *status*; the reason for sticking carefully to the latter is to be consistent. The process *state* is *all* (stateful) information associated with it, whereas the *status* is the specific item of information which details, for example, whether or not it is ready for execution.
- The scheduling queues are typically implemented as a linked list of PCBs; the process table (which includes a PCB for every process) could also be implemented as a similar list, which is sometimes termed a **job queue** or **task queue**. In addition, a pointer to the process (clearly there can only be one) currently with executing status, and hence being executed, would typically maintained: this allows direct access to said PCB, which is crucial to minimise overhead during a context switch.
- If no process is executing, and there is no process on the ready queue, what should be executed?! An OS will often solve this using a dummy, idle process designed simply to occupy the processor until a real process later becomes ready. This approach is arguably easier than a special-case in the decision process.

POSIX(ish) Realisation (5) – Creation

- A process may be created at
  - implicitly, at boot-time (e.g., `init`), *or*
  - explicitly, at run-time.

Notes:

- It is important to see that the semantics of `fork` are *one* option among a design space of options wrt.
  1. execution, i.e.,
     - the parent process waits for the child process to terminate before continuing to execute, *or*
     - the parent process continues to execute concurrently with the child process

     which can be interpreted as meaning process creation is blocking or non-blocking respectively, and
  2. address space, i.e.,
     - the child process is initialised using a duplicate of the parent address space, *or*
     - the child process is initialised with a new address space

     which you could summarise as meaning the semantics are to clone an existing process, or simple create a new one from scratch.
- Although it is a specific example, `fork` highlights some reasons that process creation is, more generally, a costly operation. An obvious reason is that (in "traditional UNIX" at least), explicitly creating a copy of the parent address space is a *huge* overhead; fortunately, it can be mitigated by techniques such as **copy-on-write** as supported by virtual memory.
- By studying the manual pages for `fork` and `exec`, e.g.,

  ```
  man exec
  ```

  it should become clear where `argc` and `argv` arguments to a standard `main` function come from: you may have (correctly) thought of these as copies of the command line argument, but given that the command line interpreter (e.g., instance of `bash`) is an executing process using `fork` to create new processes, it is in fact `exec` which loads the image including `main` and then invokes it with the arguments.
- In Windows, the `CreateProcess` system call more or less combines `fork` and `exec` into one: for each process created, it *demands* a new image is also loaded. Also in contrast to Linux, the Windows process model is st. no hierarchy is enforced: although a parent process still creates a child process, the handle held by the former for the latter can be passed freely to another process (i.e., there is no relationship and hence hierarchy enforced).

POSIX(ish) Realisation (5) – Creation

- POSIX says we need
  - `fork` [1, Page 881]:
    - create new child process with unique PID,
    - child has *copy* of parent address space and resources,
    - *both* parent and child continue to execute from call to `fork`,
    - return value is 0 for child, and PID of child for parent.
  - `exec` [1, Page 772] and friends:
    - replace current process image with new process image,
    - no return, since call point no longer exists (!)

  where a **loader** [2] performs various tasks related to the latter.

Notes:

- It is important to see that the semantics of `fork` are *one* option among a design space of options wrt.
  1. execution, i.e.,
     - the parent process waits for the child process to terminate before continuing to execute, *or*
     - the parent process continues to execute concurrently with the child process

     which can be interpreted as meaning process creation is blocking or non-blocking respectively, and
  2. address space, i.e.,
     - the child process is initialised using a duplicate of the parent address space, *or*
     - the child process is initialised with a new address space

     which you could summarise as meaning the semantics are to clone an existing process, or simple create a new one from scratch.
- Although it is a specific example, `fork` highlights some reasons that process creation is, more generally, a costly operation. An obvious reason is that (in "traditional UNIX" at least), explicitly creating a copy of the parent address space is a *huge* overhead; fortunately, it can be mitigated by techniques such as **copy-on-write** as supported by virtual memory.
- By studying the manual pages for `fork` and `exec`, e.g.,

  ```
  man exec
  ```

  it should become clear where `argc` and `argv` arguments to a standard `main` function come from: you may have (correctly) thought of these as copies of the command line argument, but given that the command line interpreter (e.g., instance of `bash`) is an executing process using `fork` to create new processes, it is in fact `exec` which loads the image including `main` and then invokes it with the arguments.
- In Windows, the `CreateProcess` system call more or less combines `fork` and `exec` into one: for each process created, it *demands* a new image is also loaded. Also in contrast to Linux, the Windows process model is st. no hierarchy is enforced: although a parent process still creates a child process, the handle held by the former for the latter can be passed freely to another process (i.e., there is no relationship and hence hierarchy enforced).

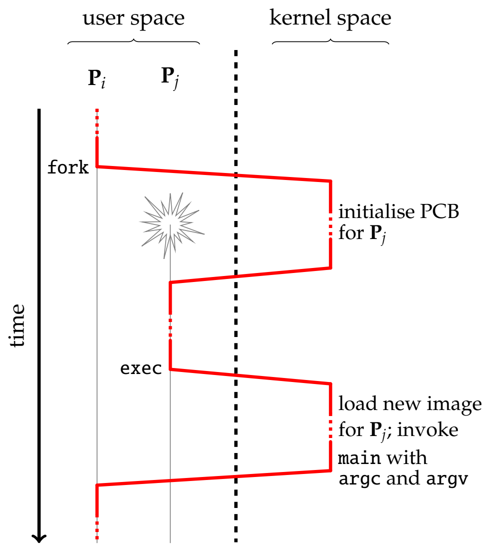
POSIX(ish) Realisation (6) – Creation

Listing (C)

```c
int main( int argc, char* argv[] ) {
  pid_t pid = fork();

  if     ( pid == 0 ) { // parent = P_i
    ...
  }
  else if( pid >  0 ) { // child  = P_j
    exec( ... );
  }
  else if( pid <  0 ) { // error
    abort();
  }

  return 0;
}
```



user space
kernel space
$\mathbf{P}_i$
$\mathbf{P}_j$
time
fork
initialise PCB for $\mathbf{P}_j$
exec
load new image for $\mathbf{P}_j$; invoke main with argc and argv

Notes:

POSIX(ish) Realisation (7) – Communication

- **Inter-Process Communication (IPC)** is a broad topic; with examples such as
  1. **message passing**
     - half- and full-duplex (anonymous) pipe,
     - named pipe (i.e., FIFO-based file),
     - socket,
     - message queue,
     - signal,
     - ...
  2. **shared memory**
     - semaphore,
     - shared memory,
     - memory-mapped file,
     - ...

Notes:

- A signal is a very simple form of IPC, which really communicates a type of (asynchronous) condition or event rather than data per se: there are a fixed set of signal types, defined in `signal.h`, one process (or the kernel) can send to another. An example is `SIGTERM`: this is used to terminate a process, so, by providing a signal handler, a process has the chance to perform any pre-termination clean-up needed (which it would otherwise be unable to). Note the `kill` command allows signalling from the command line in much the same way the `kill` function does programmatically.
- As hinted, POSIX supports a *lot* of mechanisms for IPC: in addition to the signalling case outlined here, you may want to look at
  - `pipe` [1, Page 1400],
  - `mmap` [1, Page 1309],
  - `msgget` [1, Page 1309] and friends, and
  - `shmget` [1, Page 1309] and friends

  plus `send` [1, Page 1844] and `recv` [1, Page 1759] and friends, which support socket-based communication at a larger scale (i.e., across non-local *hosts* connected to a network, rather than just local *processes* executing on such a host).

POSIX(ish) Realisation (7) – Communication

- POSIX says, *for example*, we need
  - `signal` [1, Page 1937]:
    - decide whether specific signal is handled or ignored,
    - decide whether signal is handled by default or user-defined handler, i.e.,

```
void (*handler)( int signum );
```

  - `raise` [1, Page 1731]:
    - send signal to the currently executing process.
  - `kill` [1, Page 1199]:
    - send signal to a specific process (or group thereof).

Notes:

- A signal is a very simple form of IPC, which really communicates a type of (asynchronous) condition or event rather than data per se: there are a fixed set of signal types, defined in `signal.h`, one process (or the kernel) can send to another. An example is `SIGTERM`: this is used to terminate a process, so, by providing a signal handler, a process has the chance to perform any pre-termination clean-up needed (which it would otherwise be unable to). Note the `kill` command allows signalling from the command line in much the same way the `kill` function does programmatically.
- As hinted, POSIX supports a *lot* of mechanisms for IPC: in addition to the signalling case outlined here, you may want to look at
  - `pipe` [1, Page 1400],
  - `mmap` [1, Page 1309],
  - `msgget` [1, Page 1309] and friends, and
  - `shmget` [1, Page 1309] and friends

  plus `send` [1, Page 1844] and `recv` [1, Page 1759] and friends, which support socket-based communication at a larger scale (i.e., across non-local *hosts* connected to a network, rather than just local *processes* executing on such a host).

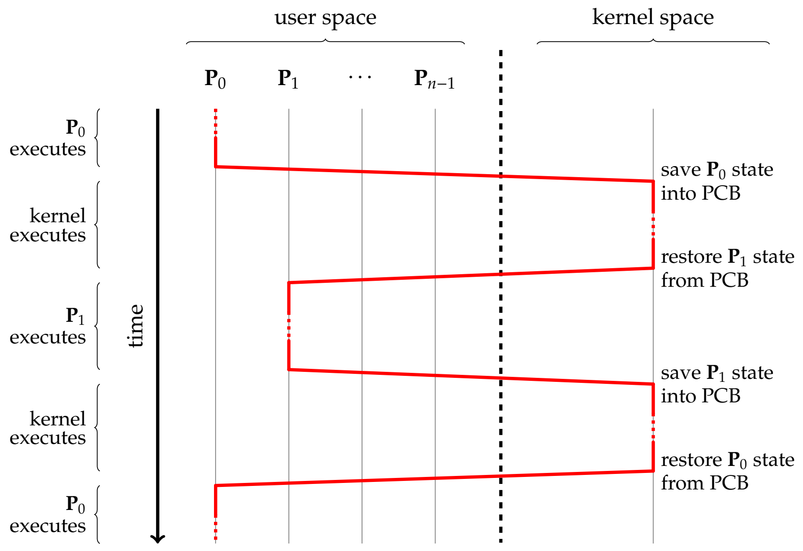
POSIX(ish) Realisation (8) – Control

- POSIX says *we* need
  - `wait` [1, Page 2181]:
    - suspend execution until process (or group thereof) terminates,
    - receive error status of said process.
  - `sleep` [1, Page 1963]:
    - suspend execution for specified time period,
    - close to, but not quite `yield`.

  plus the *kernel* needs to be able to context switch ...

Notes:

POSIX(ish) Realisation (9) – Control



Notes:

- As the diagram shows, but doesn't outright explain, efficiency of context switches is important: they represent pure overhead [3] wrt. useful execution by any of the user space processes.
- The normal definition of a context switch implicitly applies to user space processes: the idea is we suspend one process and resume another. However, one could interpret the term context more generally: an interrupt invokes a change of execution context, in the sense it causes a mode switch from user to kernel mode.
  The subtle, yet vitally important implication is that the kernel also has an execution context of sorts: this includes an address space, which, for example, houses a kernel text, data and stack segment(s) and allows the kernel to execute instructions and hence functions in the same way a user process does. As a result, *each* mode is typically initialised with an independent stack region. When the kernel is initialised, it will set the value of SP, for each mode, to some address st. that mode can create and unwind stack frames just like a "normal" program.
- The diagram illustrates the fact that realising each context switch needs two mode switches (to enter and leave kernel mode), but also performs some additional steps (e.g., saving and restoring state to and from PCBs). It *must* be realised in kernel mode, because only the kernel is able to access *all* the state (including any protected resources) associated with each process. A mode and context switch *are* related, but also different concepts: the former relates more to hardware (the processor) while the latter relates more to software (the processes).

POSIX(ish) Realisation (10) – Termination

- A process may terminate due to
  - exit,
  - controlled error,
  - uncontrolled error, or
  - signal

  which can be classified as normal, abnormal and external events.

Notes:

- The concepts of orphaned and zombie processes both relate to semantics of process termination. Note, for example, that
  - Some kernels prevent orphaned processes from existing by enforcing application of **cascading termination**: if the parent terminates, the (sub-)tree of child processes are implicitly terminated as well.
  - Linux in fact allows orphaned processes, in the sense it forces the `init` process to "adopt" them.
  - When a process terminates via `exit`, it can return an exit status (or code) to the parent; the parent process receives this via `wait`. The child process PCB must be retained until the parent invokes `wait`, because the PCB houses associate state *including* the exit status. So if the parent fails to do so, the child is a zombie.
- Returning from `main` (which, recall, has an `int` return type normally) has the same semantics as controlled termination using `exit`. The `EXIT_SUCCESS` and `EXIT_FAILURE` constants defined `stdlib.h` (symbolically) represent two obvious exit statuses, but a zero (resp. non-zero) are more generally interpreted as success (resp. failure).
- Whenever a process terminates due to a signal, the kernel will manage the signal-based IPC mechanism so *it* terminates the process. The signal *source* may be another process *or* the kernel itself, however: the former depends on the source process having permission to signal the target. Although a range of situations might prompt the OS to terminate a process (which hasn't caused an error per se), the Linux Out-Of-Memory (OOM) mechanism, an overview of which is presented by

  `http://linux-mm.org/OOM_Killer`

  is a good example.

POSIX(ish) Realisation (10) – Termination

- POSIX says we need
  - `exit` [1, Page 785]:
    - perform normal termination,
    - invoke call-backs, flush then close open files
    - pass exit status to parent process (via `wait`).
  - `abort` [1, Page 556]:
    - perform abnormal termination.

  where, in both cases, the associated PCB is (eventually) removed.

Notes:

- The concepts of orphaned and zombie processes both relate to semantics of process termination. Note, for example, that
  - Some kernels prevent orphaned processes from existing by enforcing application of **cascading termination**: if the parent terminates, the (sub-)tree of child processes are implicitly terminated as well.
  - Linux in fact allows orphaned processes, in the sense it forces the `init` process to "adopt" them.
  - When a process terminates via `exit`, it can return an exit status (or code) to the parent; the parent process receives this via `wait`. The child process PCB must be retained until the parent invokes `wait`, because the PCB houses associate state *including* the exit status. So if the parent fails to do so, the child is a zombie.
- Returning from `main` (which, recall, has an `int` return type normally) has the same semantics as controlled termination using `exit`. The `EXIT_SUCCESS` and `EXIT_FAILURE` constants defined `stdlib.h` (symbolically) represent two obvious exit statuses, but a zero (resp. non-zero) are more generally interpreted as success (resp. failure).
- Whenever a process terminates due to a signal, the kernel will manage the signal-based IPC mechanism so *it* terminates the process. The signal *source* may be another process *or* the kernel itself, however: the former depends on the source process having permission to signal the target. Although a range of situations might prompt the OS to terminate a process (which hasn't caused an error per se), the Linux Out-Of-Memory (OOM) mechanism, an overview of which is presented by

  `http://linux-mm.org/OOM_Killer`

  is a good example.

An Aside: Threads vs. Processes

Definition (user thread vs. kernel thread)

If a process has $n > 1$ threads, it is said to be **multi-threaded**; those threads might be realised in

- user space (viz. a **user thread**) yielding an n-to-1 mapping, or in
- kernel space (viz. a **kernel thread**) yielding a 1-to-1 mapping, or
- a hybrid, yielding an n-to-m mapping

from threads to kernel-exposed **schedulable entities**.

Definition (software thread vs. hardware thread)

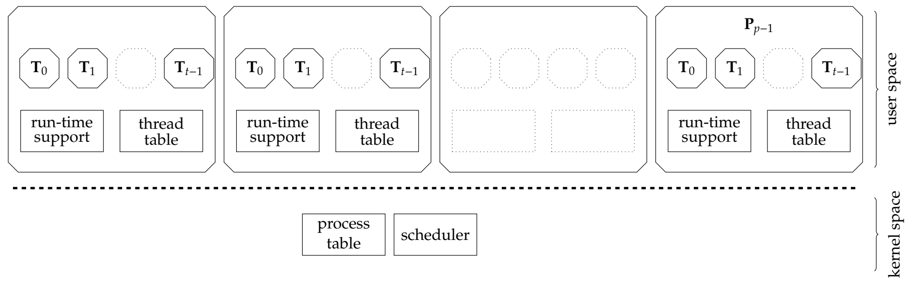
A **software thread** (resp. **hardware thread**) uses software (resp. hardware) resources to capture each execution context.

Notes:

- Fundamentally, the difference between user or kernel threads boils down to the definition of what a schedulable entity is. If the kernel "knows about" threads (i.e., they are realised in kernel space), then threads are the entity controlled by the scheduler. If not (i.e., they are realised in user space), then processes are the entity controlled by the scheduler: the user threads are managed in user space, with the kernel oblivious of this fact.
- Ungerer et al. [?] present a good survey of processor design options relating to hardware multi-threading. Use of such an approach can be summarised as allowing concurrent execution of instructions from various *different* threads using *one* execution pipeline; this suggests support for multiple register files in hardware (on per thread).

An Aside: Threads vs. Processes – User vs. Kernel Threads

- Roughly per [7, Sections 2.24-2.26]



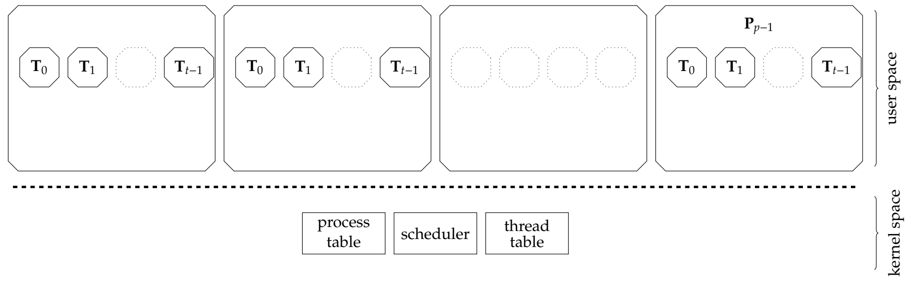
one can identify advantages for both

- user thread implementation, e.g.,
  - can port to a kernel *without* thread support,
  - can implement process-specific policies,
  - lower-overhead creation and context switch,
  - *all* threads are blocked by blocking system call
  - ...

Notes:

An Aside: Threads vs. Processes – User vs. Kernel Threads

- Roughly per [7, Sections 2.24-2.26]



one can identify advantages for both

- kernel thread implementation, e.g.,
  - blocking system calls are thread-specific,
  - easier to cope with interrupts,
  - `kill` and `fork` need different semantics.
  - ...

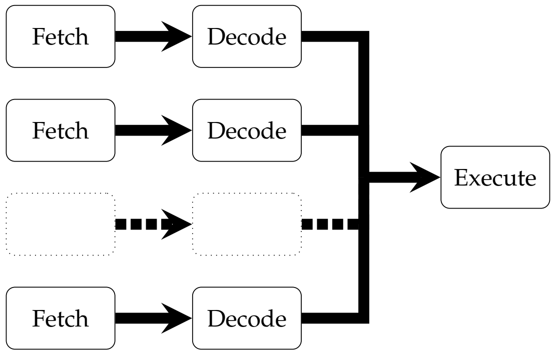
Notes:

- Most implementations of x86-32 and x86-64 are
  - *deeply* pipelined, and/or
  - *highly* superscalar,

  for example the Pentium 4 has roughly 20 pipeline stages ...
- .. this has (at least) two implications: the
  1. demand for instructions to execute is high, and
  2. cost of a pipeline stall or flush is high.

Notes:

- Idea: **Simultaneous MultiThreading (SMT)**, or Intel Hyper-Threading (e.g., for Pentium 4):



i.e.,

- replicate parts of the processor that house execution context, and thus
- improve utilisation of (later) execution unit(s).

Notes:

- From an architectural viewpoint, SMT gives some clear advantages. In particular, a superscalar design suffers acutely from the von Neumann bottleneck: the impact of memory latency makes it hard to keep the execution units at a high level of utilisation. Allowing two threads to be interleaved potentially allows a better instruction mix, and so the potential of latency hiding (e.g., by one compute-bound thread of another memory-bound thread).

Conclusions

Continued in next lecture ...

Notes:

References

[1] Standard for information technology - portable operating system interface (POSIX).
Institute of Electrical and Electronics Engineers (IEEE) 1003.1, 2008.
http://standards.ieee.org/findstds/standard/1003.1-2008.html.

[2] J.R. Levine.
Linkers & Loaders.
Morgan-Kaufmann, 2000.

[3] C. Li, C. Ding, and K. Shen.
Quantifying the cost of context switch.
In *Experimental Computer Science (ExpCS)*, number 2, 2007.

[4] A. Silberschatz, P.B. Galvin, and G. Gagne.
Chapter 3: Process concept.
In *Operating System Concepts* [5].

[5] A. Silberschatz, P.B. Galvin, and G. Gagne.
Operating System Concepts.
Wiley, 9th edition, 2014.

[6] A.S. Tanenbaum and H. Bos.
Chapter 2.1: Processes.
In *Modern Operating Systems* [7].

Notes:

References

[7] A.S. Tanenbaum and H. Bos.
Modern Operating Systems.
Pearson, 4th edition, 2015.

Notes: