Operating Systems

Tutorial 1

Michael Tänzer

`os-tut@nhng.de`
`http://os-tut.nhng.de`

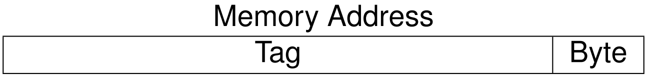
9th November 2010

Outline

True or False

- Reading only from a user provided buffer is save
- The interrupt vector is a bit field that enables or disables the delivery of the different interrupt types
- Multi-programming only makes sense when doing interrupt driven I/O
- One must use kernel stacks in multi-processor environments in order to protect the kernel

Fully Associative



- Each memory block may be mapped to each cache line

+ Ideal utilisation of the given space

– Lots of comparators needed (one per cache line) $\Rightarrow$ expensive

Direct Mapped

Memory Address

Tag	Cache Line	Byte

- Each memory block is mapped on the cache line corresponding to the lower bits of its address (without the bits used within each block)

+ Only one comparator needed

– If more than one memory block that are mapped to the same cache line are in use they replace each other $\Rightarrow$ cache useless

n-way Set Associative

Memory Address

Tag	Set	Byte

- Compromise of the other two
- Each memory block is mapped to a subset of *n* cache lines (like the direct mapped approach) inside these sets it's mapped like in the fully associative organisation

+ *n* comparators needed

+ It takes more than *n* memory blocks mapped to the same set to cause the blocks replacing each other

- $n = 1 \Rightarrow$ direct mapped, $n = m \Rightarrow$ fully associative

Principle and Benefits

- High speed memory is expensive
- Only the currently needed data is kept in high-speed memory
- If there is no space left for new data, data that hasn't been used for a long time is moved down in the hierarchy
- This is done recursively for L1, L2, (L3), RAM and hard disk
- Most of the time data can be found in high speed memory without a costly pure high-speed system

What typical behaviour is utilised?

Temporal Locality	Data that has been accessed recently is likely to be accessed again (loops, etc.)
Spatial locality	Data that is close together is likely to be accessed successively (arrays, etc.)

Example: Stack

Why is the cache divided into cache lines?

What might limit the size of a cache line?

- Transferring big parts at one time (burst mode) is faster than transferring multiple little units
- Transferring parts that are never used is unneeded effort
- $\Rightarrow$ need to find the right balance

What keeps a process from accessing the memory contents of another process?

- Each process has its own view on the memory (address space)

⇒ The same (virtual) address will mostly refer to different physical memory locations in different processes

- A process can’t access memory of another process unless they agree to share some data
- Protection is implicit: if you can’t name it you can’t touch it
- Mapping from virtual to physical memory addresses (address translation) is done by hardware (MMU) and the OS

Why was time sharing not widespread on early so-called ‘batched’ systems?

- No I/O terminals, only punch cards or magnetic tapes
- Early systems didn’t provide a multiprogramming environment
- No hardware support for address spaces

What is a Virtual Machine (VM)?

- Examples: JVM, VMWare, VirtualBox, XEN, …
- A simulator (called VM monitor) which runs on machine *A* provides a simulation of machine *B* (called virtual machine).
- *B* doesn't need to be a 'real' machine (e. g. JVM)
- If $B \approx A$ many parts may be executed natively $\Rightarrow$ little overhead
- If *B* is as complex but very different from *A* simulation might be difficult $\Rightarrow$ large simulator, significant overhead

How to allow a guest OS to securely execute privileged instructions?

Paravirtualisation

- Modify guest OS replacing privileged instructions with calls to VM monitor / hypervisor
  - Need source code of the guest OS
  - Has to be done manually
  - If the guest OS changes the patches have to be ported to the new version

How to allow a guest OS to securely execute privileged instructions?

Binary Translation

- Hypervisor scans the machine code of the guest OS and dynamically replaces privileged instructions with calls to the hypervisor or some emulation code
- + No need to modify guest or host OS
- – Very tricky to get right

How to allow a guest OS to securely execute privileged instructions?

Trap-and-Emulate

- Whenever the guest OS executes a privileged instruction an exception occurs which is handled by the hypervisor

– Might need support in the kernel

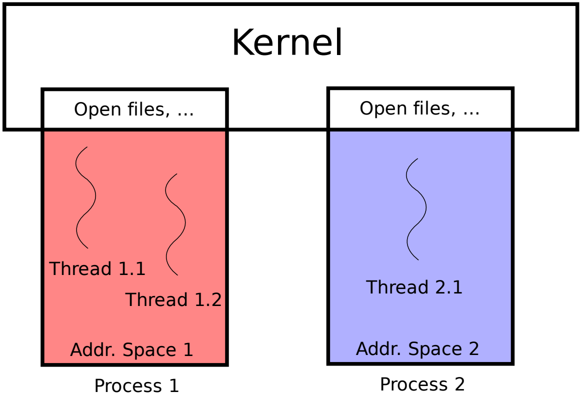
– Not possible for x86 (without special hardware support)

+ No need to modify guest OS

+ Hardware support available in all modern x86 CPUs

What are threads, processes and address spaces?

How are they related?



Process vs. Programme

Programme	Specifies the rules of an execution
Process	Describes a particular execution of this programme including all runtime state:

- Registers
- Memory
- Open files
- …

Depict the possible state transitions and the events that cause them

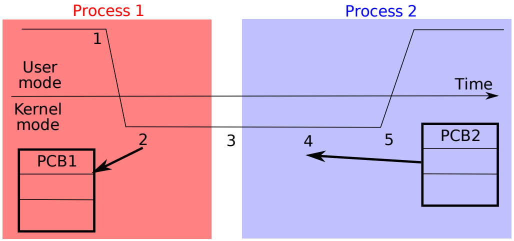
Assume an OS that supports the following process states:

- New
- Running
- Ready
- Waiting
- Terminated

What does a process control block (PCB) contain?

- Process state
- Process ID (PID)
- Program counter
- Registers (including stack pointer)
- Credentials (UID, GID)
- Exit status
- Accounting, scheduling, memory management and I/O status information
- Most entries are only valid if the process is currently not running

Describe how the kernel performs a context switch



1. Interrupt/exception/syscall occurs
2. Save execution state of the interrupted/preempted process/thread to its PCB
3. Maybe switch address spaces, flush caches etc.
4. Load execution state of process/thread to be run next from its PCB
5. Return to user mode

CPU- vs. I/O-bound processes

- CPU-bound processes rarely do I/O

$\Rightarrow$ They are unlikely to block

- I/O-bound only perform short CPU bursts between doing I/O

$\Rightarrow$ They are likely to block very often

What does the `fork` syscall do?

- Create a child process which is identical to the parent process
    - Same memory contents
    - Same open files
    - Same register contents (including program counter)
    - ...
- Exceptions:
    - Different process ID (PID)
    - Different address space (contents are the same though)
    - ...
- Returns child's PID to the parent and 0 to the child

Compare `CreateProcess` with `fork`

Strengths and Weaknesses

- `CreateProcess` ≈ `fork` + `execve`
- + Saves one system call
- – Less flexible (can't use processes like some kind of pseudo-threads)
- – Modifying the context of the new process is more complex
  - In UNIX you just `fork` change the context and then do the `execve` (e. g. redirect `stdin/stdout`)
  - In Windows you have to build a `STARTUPINFO` structure and pass a pointer to it to `CreateProcess`

Example

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main(int argc, char **argv){
    /* Parent prints "I am the parent!\n",
     *  "Child PID is %d\n", "Child terminated\n"
     *
     * Child prints "I am the child!\n"
     *
     * waitpid(pid, NULL, 0) will wait for the
     * child with the specified PID to exit
     */
    return 0;
}
```

Example

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main(int argc, char **argv){
    printf("I am the parent!\n");
    pid_t pid = fork();

    if (pid > 0){
        printf("Child PID is %d\n", pid);
        waitpid(pid, NULL, 0);
        printf("Child terminated\n");
    }
    else if (pid == 0){
        printf("I am the child!\n");
    }
    else{
        printf("Error. Fork failed\n");
    }

    return 0;
}
```

Is `fork` sufficient to implement a shell?

- `fork` allows us to create a new process
- We need a way to replace the code of the current process with the one that shall be executed
- `execve` does that
- With `execve` the shell can
  1. Create a new process (`fork`)
  2. Child: Possibly make some adjustments (e. g. redirect standard output)
  3. Child: Replace the code of the shell with the code of the programme to be run (`execve`)
  4. Parent: Wait for the programme to terminate (`wait`)
  5. Show prompt

Questions & Comments

Any questions or comments?

Politically Correct UNIX

System VI Release notes

- `man` pages are now called `person` pages
- Similarly `hangman` is now the `person_executed_by_an_oppressive_regime`
- The `more` command reflects the materialistic philosophy of the Reagan era. System VI uses the environmentally preferable `less` command
- There will no longer be a invidious distinction between ‘dumb’ and ‘smart’ terminals. All terminals are equally valuable
- No longer will it be permissible for files and processes to be ‘owned’ by users. All files and processes will own themselves, and decide how (or whether) to respond to requests from users.