This work is licensed under a Creative Commons license

Attribution-NonCommercial-NoDerivatives 4.0 International
(CC BY-NC-ND 4.0)

You are free to:

Share copy and redistribute the material in any medium or format.

Under the following terms:

Attribution You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

NonCommercial You may not use the material for commercial purposes.

NoDerivatives If you remix, transform, or build upon the material, you may not distribute the modified material.

Introduzione ai sistemi operativi

(all'interno del corso SETI)

Giovanni Lagorio

giovanni.lagorio@unige.it
https://csec.it/people/giovanni_lagorio
Twitter & GitHub: zxgio

DIBRIS - Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi
University of Genova, Italy

www.zenhack.it

Outline

Introduzione

- `whoami`
- Come anticipato, per la parte pratica useremo Linux
- Per le demo userò Ubuntu 22.04 LTS
  - qualsiasi distro va bene
  - WSL1 no, WSL2 dovrebbe. . . in dubbio, usate una VM

Libro di testo

Il testo principale è Operating Systems: Three Easy Pieces
`http://pages.cs.wisc.edu/~remzi/OSTEP/`

- in inglese
- gratuito e fatto *molto bene*
- i tre "pieces" sono virtualizzazione, concorrenza e persistenza
  - non copriremo tutto OSTEP e, viceversa, ci servirà materiale aggiuntivo
  - da poco sono disponibili anche capitoli su sicurezza (di un altro autore)
- link ai capitoli, e non i PDF, perché gli autori li tengono aggiornati
- queste prime slide sono basate su:
  `http://pages.cs.wisc.edu/~remzi/OSTEP/intro.pdf`

Outline

Cosa fa “girare” i programmi?

- CPU: fetch (da PC/IP), decode ed execute
  - all’inizio, faremo finta di avere una sola CPU con un solo *core*
  - come facciamo a sapere se è successo qualcosa nel mondo esterno?
- quanti programmi per volta?
- chi porta i programmi in RAM?
- quando avete scritto dei programmi, vi siete preoccupati dell’ambiente circostante? Altri programmi, hardware, etc?

Gestire e virtualizzare

- un software particolare, il (*kernel*/nucleo del) sistema operativo, si occupa di gestire e virtualizzare le risorse come CPU, memoria, ...
  - ogni processo "crede" di avere (almeno) una CPU e memoria tutta sua
  - problemi: efficienza e *fairness* (equità)
- e il resto dell'HW? L'I/O?
  - accessi concorrenti/protocolli diversi $\Rightarrow$ *device-driver*
  - interfacce "scomode" $\Rightarrow$ *file-system*
    - "The nice thing about standards is that there are so many of them to choose from — Andrew S. Tanenbaum" $\Rightarrow$ VFS

Una “macchina virtuale”

In un certo senso, per ogni processo, il s.o. crea una macchina virtuale:

- con le sue CPU, la sua memoria, . . .
  → `c-examples/cpu_example.c`
  → `c-examples/mem_example.c`
- alcune istruzioni vengono eseguite direttamente dalla/e vera/e CPU
  - ma vedremo che non tutte le istruzioni saranno disponibili
- il s.o. esporta delle API (le chiamate di sistema) che possiamo vedere come “istruzioni speciali” di questa VM

Problema: come si virtualizzano in modo efficiente le risorse? Serve del supporto HW?

Obiettivi progettuali

- virtualizzare le risorse HW minimizzando l'*overhead*
- fornire protezione e isolamento fra processi diversi (e il s.o. stesso)
  - meccanismi di sicurezza; per esempio, per preservare confidenzialità/integrità di dati di utenti diversi
- il s.o. deve essere robusto e affidabile
  - che conseguenze potrebbero avere dei bug nel kernel?

Outline

Un po' di storia

- primissimi sistemi, *batch processing*: s.o. == libreria; la scelta del *batch* da eseguire sul *mainframe* era fatta dall'operatore (umano)
  - problema: protezione — il codice del s.o. dovrebbe essere "speciale"
  - servono due modalità di esecuzione distinte: sistema/kernel vs utente
  - un meccanismo per passare da una all'altra
  - system call (AKA syscall) analoga a una chiamata a funzione, ma con "qualcosa" in più
  - tipicamente implementata tramite una *trap*/eccezione/interrupt "SW"
    - lo stesso meccanismo per gestire, per esempio, divisioni per 0
    - su x86 abbiamo `INT` n (ma anche `SYSENTER` e `SYSCALL`)
  - trap vs interrupt == eventi sincroni vs eventi asincroni
  - si utilizzerà un'istruzione speciale per restituire il controllo (ripristinando la modalità precedente)
    - su x86: `IRET`

Multiprogrammazione

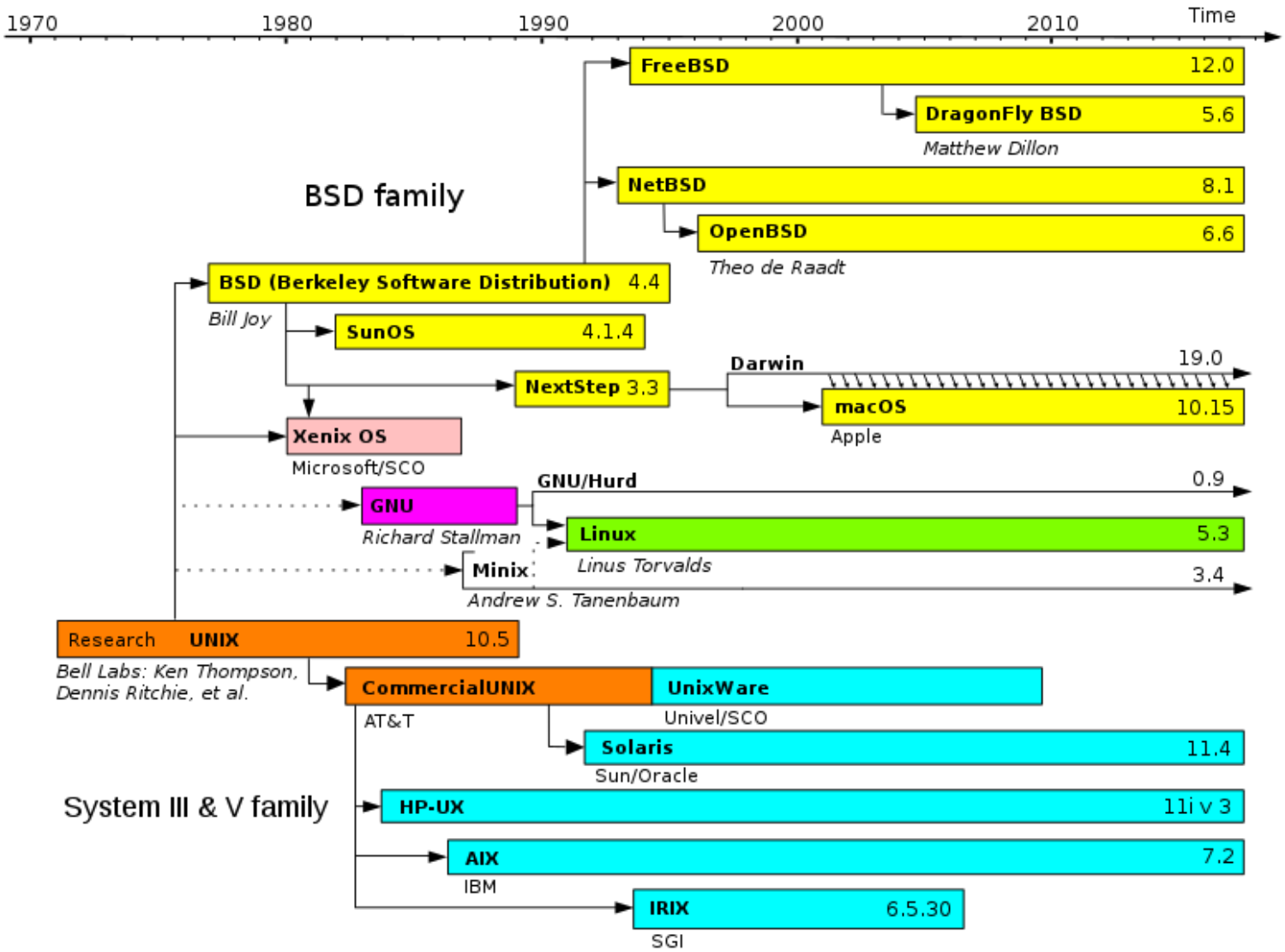
- avvento dei minicomputer
- idea di sfruttare i tempi di I/O per fare dell'altro: più *job* in memoria allo stesso tempo, alternando la CPU fra di loro $\rightarrow$ multiprogrammazione
  - problemi da affrontare: protezione e concorrenza

Outline

Unix: nascita

- Unix è un sistema operativo nato nei Bell Labs di AT&T a metà degli anni '60 (`https://en.wikipedia.org/wiki/History_of_Unix`)
  - potete trovare le edizioni storiche dei *Programmer's manual* su github: `https://github.com/dspinellis/unix-v4man`
- al tempo stesso semplice e potente
- scritto in C (e assembler)
- sorgenti distribuiti praticamente gratis alle università, fino al 1984
  - licenza $20.000 vs $150/200, a seconda delle fonti
- poi "fork", e.s. BSD, varie versioni commerciali, tutte più o meno (in)compatibili fra loro
- oggi, marchio registrato; solo sistemi certificati possono usare nome
- vari Unix-like open-source: Linux, FreeBSD, NetBSD, xv6, ...

Unix: fork



https://commons.wikimedia.org/wiki/File:Unix_timeline.en.svg

Standard

- verso la fine degli anni '80 tentativi di creare standard per interoperabilità
- POSIX (Portable Operating System Interface), di IEEE, e Single UNIX Specification di The Open Group
  - nel 2008, fusi in "Open Group Base Specification"
- le attuali specifiche SUS sono *contemporaneamente* standard POSIX e ISO/IEC 9945
  `https://pubs.opengroup.org/onlinepubs/9699919799/`
  `https://publications.opengroup.org/t101`
- nessuno dei sistemi che useremo è POSIX al 100%, ma sono tutti *POSIX-oriented*
  `en.wikipedia.org/wiki/POSIX#POSIX-oriented_operating_systems`
- in questo corso non parleremo mai di Unix™: tutte le volte che parleremo di Unix sarà sottinteso Unix-*like*

Linux vs Xv6

- per le esercitazioni useremo Linux
- chi vuole approfondire può dare un'occhiata a un nostro *fork* di Xv6
  `https://github.com/zxgio/xv6-SETI`
  un sistema operativo *didattico* sviluppato al MIT
  progetto originale:
  `https://pdos.csail.mit.edu/6.828/2018/xv6.html`
- se Linux è open-source, perché tiriamo in ballo Xv6?
  - nel 2020, i sorgenti del kernel di Linux erano quasi 28 milioni di righe di codice, in (circa) 28 mila file C
  - il kernel di Xv6 sono meno di 5 mila righe di C, in 26 sorgenti