Concurrency Models

* actual implementations * (not abstract/theoretical models)

- threads:
  - shared memory space
    - multiple stacks \(\rightarrow\) 1 per thread of control
    - concurrent programming model
      - (multiple points in code are being executed at once)

  2 ways to implement:

  user level vs. kernel level

  (Linux \(\rightarrow\) clone system call)

- process:
  - separate address space
  - separate OS state (e.g., fds)

- virtual machine:
  - separate OS
    - "physical" memory & devices
**Performance:** events vs threads

- memory size: 64k - 100k = 64KB memory

64KB stack x 100k threads

- memory footprint
  - cache locality (or not)
- scheduling
  - $O(\log(n))$ to $O(n)$
  - (now ~ $O(1)$)

Protothreads (Dentals 06)

```plaintext
PT-BEGIN
while (true)
  radio.OK()
  timer = tsleep
  if not (done())
    wait_timer = X
  PT-WAIT-UNTIL (done() or expired(wait_timer))
  ...
PT-END
```
Tame (USENIX 07)

e
- tvars a, b, c

- contrôle the variables (no longer on stack)

↓

- non-recursive - variables stored in well known locations

Capriccio (U Waterloo)

Stack

Link stack

- Allocate chunks as needed for the stack

Code

Stack check
Kernel vs User threads

User threads:
- Performance advantages (no user to kernel space switch)

Kernel threads:
- SMP \rightarrow \text{Simultaneous Multi-processing}
- OS Blocking

Scheduler activation: (NetBSD) (UW 1992)

* in between user & kernel threads

3 threads:

- User scheduler knows about block

Problem: Requires major surgery in the kernel.

Thread blocks
light weight threads / non-thread substitutes

Driving forces:
- Large servers (e.g. 100K simultaneous connections)
- Very small systems
  - TinyOS

\[ A \quad B \quad C \quad \ldots \quad \Rightarrow \quad A() \quad next = B \quad B() \quad next = C \]

Button Handler \( c_3 \)
Send Pkt()
Pkt Error Handler \( c_3 \)

Problem: Difficult to program & debug w/ events.