GETTING INFO ABOUT PROCESSES...

/proc

"virtual filesystem"

cat /proc/cpuinfo

meminfo

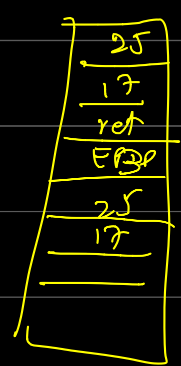
interrupts

ls -l /proc/1

cat /proc/1/maps

status

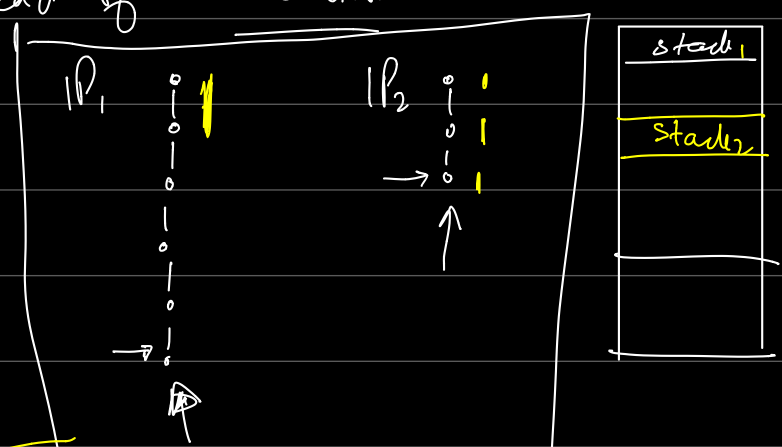
/*/status

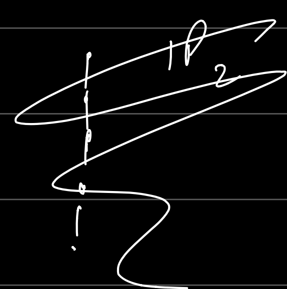


x ——— x

"Light weight processes"

" Thread of Execution"





Supervisor

Operating Systems Design

5. Threads

Paul Krzyzanowski

pxk@cs.rutgers.edu

Thread of execution

Single sequence of instructions

- Pointed to by the program counter (PC)
- Executed by the processor

Conventional programming model & OS structure:

- Single threaded
- One process = one thread

Multi-threaded model

A thread is a subset of a process:

– A process contains one or more ~~kernel~~ threads

Share memory and open files

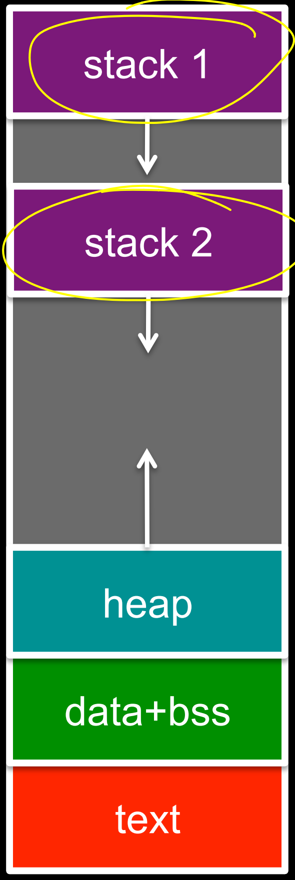
– BUT:
separate program counter, registers, and stack

– Shared memory includes the heap and global/ static data

– No memory protection among the threads

Preemptive multitasking:

– Operating system preempts & schedules threads



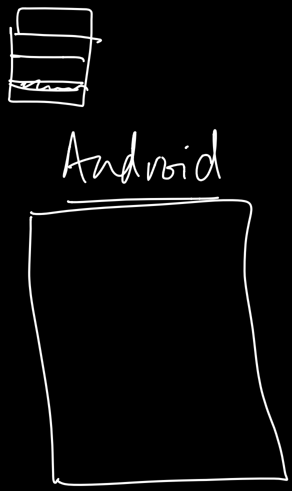
concurrent : actually in parallel

simultaneous : illusion of concurrency

Sharing

Threads share:

- Text segment (instructions)
- Data segment (static and global data)
- BSS segment (uninitialized data)
- Open file descriptors
- Signals
- Current working directory
- User and group IDs



Threads do not share:

- Thread ID
- Saved registers, stack pointer, instruction pointer
- Stack (local variables, temporary variables, return addresses)
- Signal mask
- Priority (scheduling information)

Why is this good?

Threads are more efficient

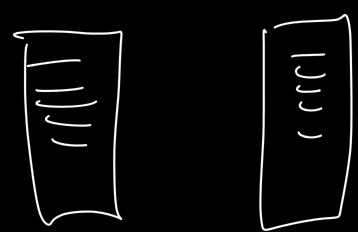
- Much less overhead to create: no need to create new copy of memory space, file descriptors, etc.

Sharing memory is easy (automatic)

- No need to figure out inter-process communication mechanisms

Take advantage of multiple CPUs – just like processes

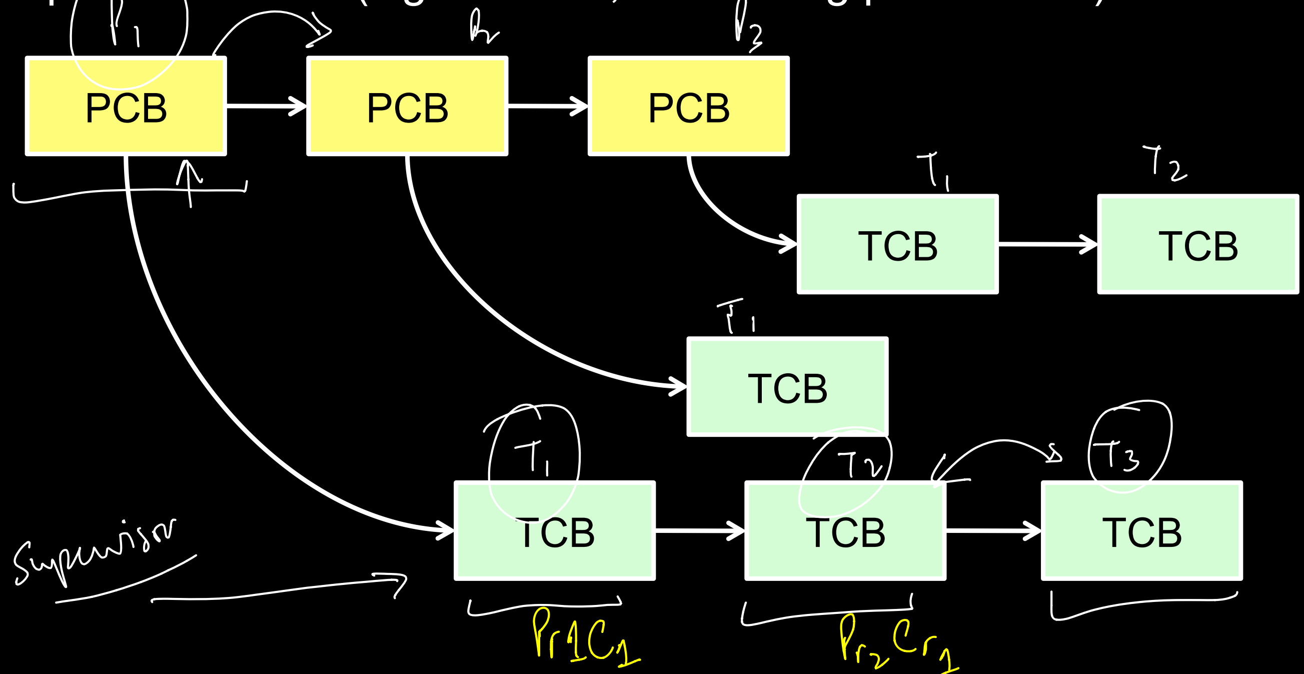
- Program scales with increasing # of CPUs
- Take advantage of multiple cores



Implementation

Process info (Process Control Block) contains one or more Thread Control Blocks (TCB):

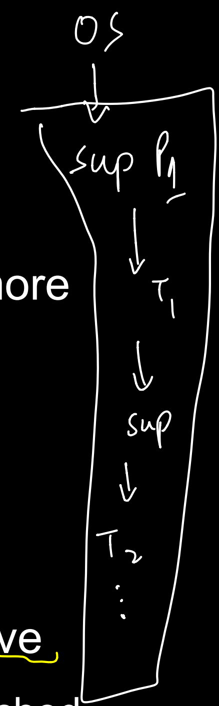
- Thread ID
- Saved registers
- Other per-thread info (signal mask, scheduling parameters)



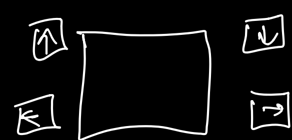
Scheduling

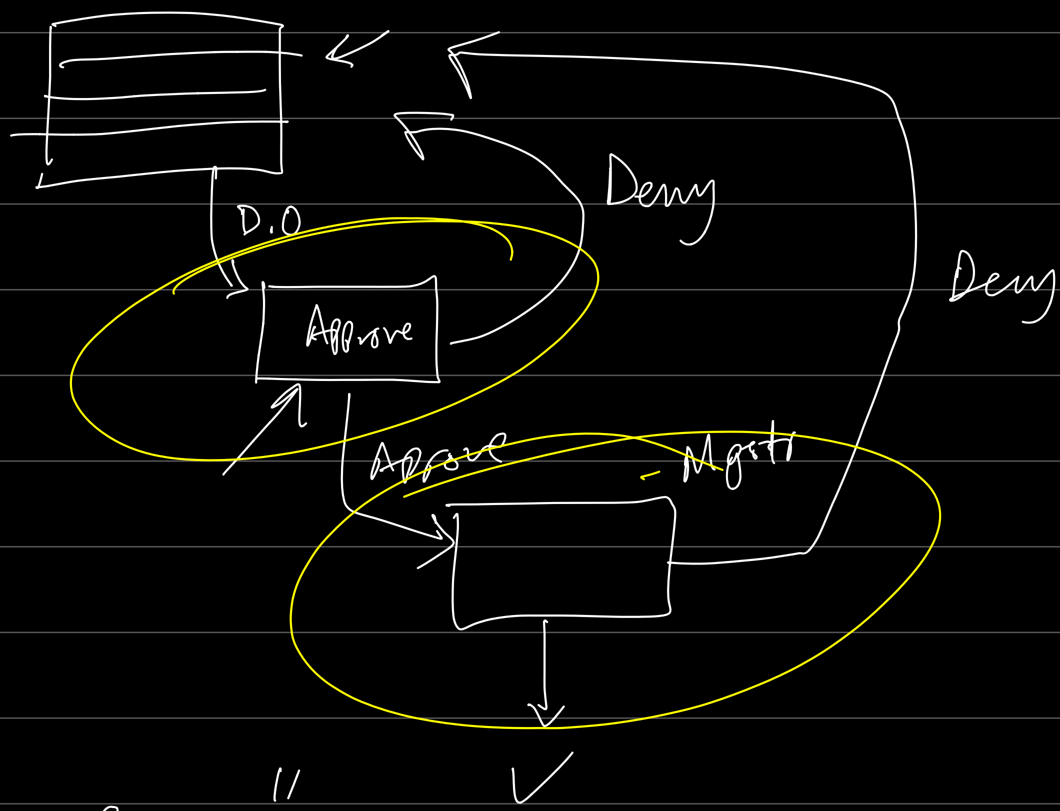
Supervisor

- A threaded-aware operating system scheduler schedules *threads*, not *processes*
  – A process is just a container for one or more threads



- Scheduler has to realize:
  – Context switch among threads of different processes is more expensive:
    - Flush cache memory (or have memory with process tags)
    - Flush virtual memory TLB (or have tagged TLB)
    - Replace page table pointer in memory management unit
  – Scheduling threads onto a different CPU is more expensive
    - The CPU's cache may have memory used by the thread cached
    - CPU affinity





"Work flow"

SAP

User requirements → Flows

Code → process

– Business Process

↳ Tasks

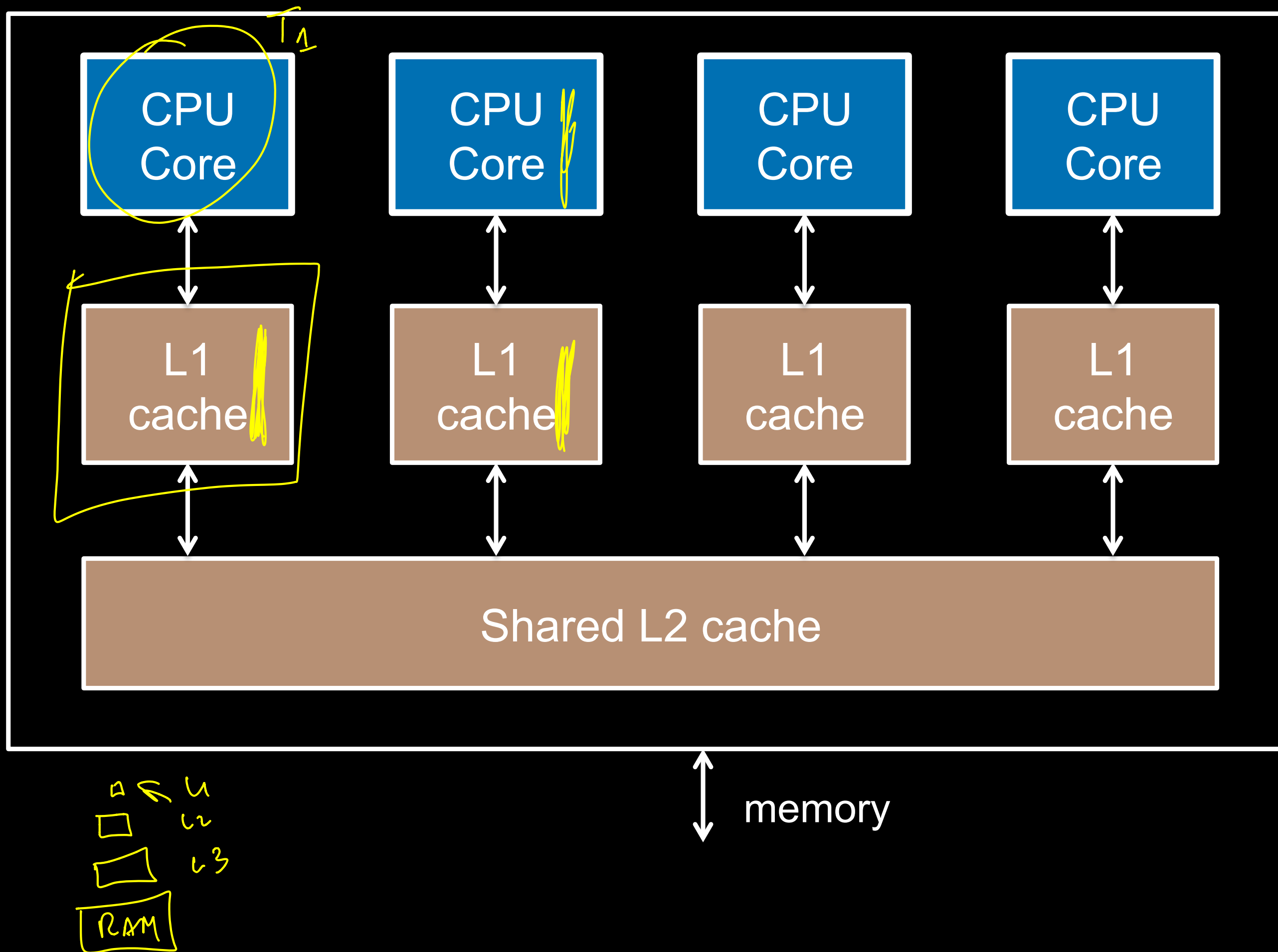
Instances → Thread

" →

" →

Multi-core architecture

CPU affinity



Programming patterns

Design Patterns

→ Programming Patterns.

HTTP 1.1

HTTP 2.0

– message passing.

Single task thread

create – serve – die

– Do a specific job and then release the thread

Worker threads

create – serve – wait

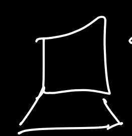
→ specify

– Specific task for each worker thread
– Dispatch task to the thread that handles it

Thread pools

create – serve – wait ←

– Create a pool of threads *a priori*
– Use an existing thread to perform a task; wait if no threads available
– Common model for servers



When are threads created?
What do they do?

Kernel-level threads vs. User-level threads

Kernel-level

- Threads supported by operating system
- OS handles scheduling, creation, synchronization

User-level

- Library with code for creation, termination, scheduling
- Kernel sees one execution context: one process
- May or may not be preemptive

Userspace

kernel space

User-level threads

Advantages

- Low-cost: user level operations that do not require switching to the kernel
- Scheduling algorithms can be replaced easily & custom to app
- Greater portability

Disadvantages

- If a thread is blocked, all threads for the process are blocked
  - Every system call needs an asynchronous counterpart
- Cannot take advantage of multiprocessing

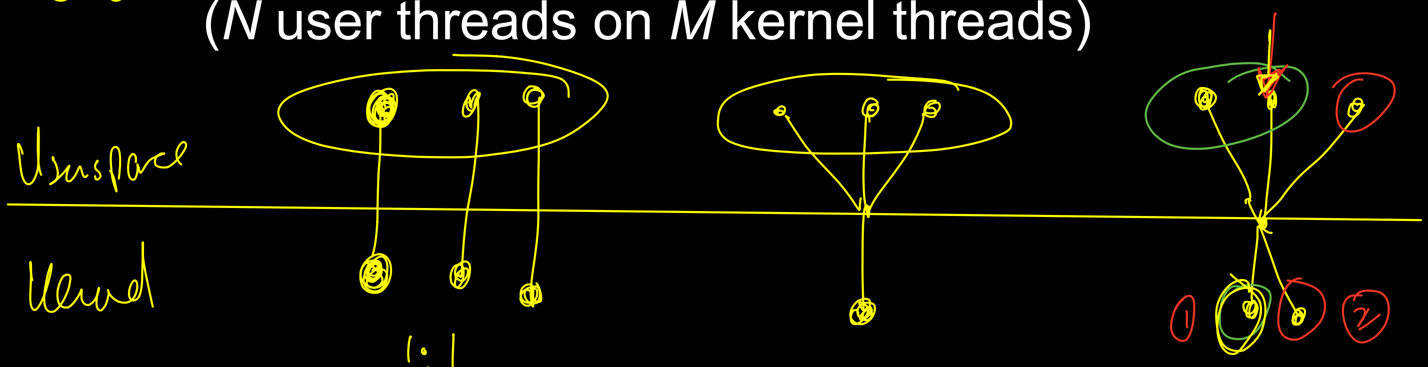
You can have both

User-level thread library on top of multiple kernel threads

1:1 – pure kernel threads only
(1 user thread = 1 kernel thread

N:1 – pure user threads only
(*N* user threads on *1* kernel thread/process)

N:M – hybrid threading
(*N* user threads on *M* kernel threads)



pthreads: POSIX Threads

- POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995)
- Defines API for managing threads
- Linux: native POSIX Thread Library (as of 2.6 kernel)
- Also on Solaris, Mac OS X, NetBSD, FreeBSD
- API library on top of Win32

"pthread.h not found"

Using POSIX Threads

Create a thread

struct

```
pthread_t t;
pthread_create(&t, NULL, func, arg)
```

function to execute

job/task/service.

~~fork()~~

- Create new thread *t*
- Start executing function $\boxed{\textit{func(arg)}}$

Join two threads:

```
void *ret_val;
pthread_join(t, &ret_val);
```

returned value

(wait)

- Wait for thread *t* to terminate (via *return* or *pthread_exit*)

No parent/child relationship!

- Any one thread may wait (join) on another thread

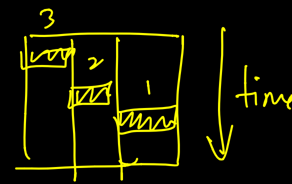
Linux *clone()* system call

- Clone a process, like *fork,* but:
  - Specify function that the child will run (with argument)
    - Child terminates when the function returns
  - Specify location of the stack for the child
  - Specify what's shared:
    - Share memory (otherwise memory writes use new memory)
    - Share open file descriptor table
    - Share the same parent
    - Share root directory, current directory, and permissions mask
    - Share namespace (mount points creating a directory hierarchy)
    - Share signals
    - *And more…*
- ~~Used by pthreads~~

Threading in hardware

(Not included)

- Hyper-Threading (HT) vs. Multi-core vs. Multi-processor
- One core = One CPU
- Hyper-Threading
  - One physical core *appears* to have multiple processors
    - Looks like multiple CPUs to the OS
  - Multiple threads run but compete for execution unit
  - Events in the pipeline switch between the streams
  - Threads do not have to belong to the same process
    - But the processors share the same cache
    - Performance can degrade if two threads compete for the cache
  - Works well with instruction streams that have large memory latencies



"Amdahl's Law"

100000,00

100s

500,000

500000

59s

100000

2s

The End