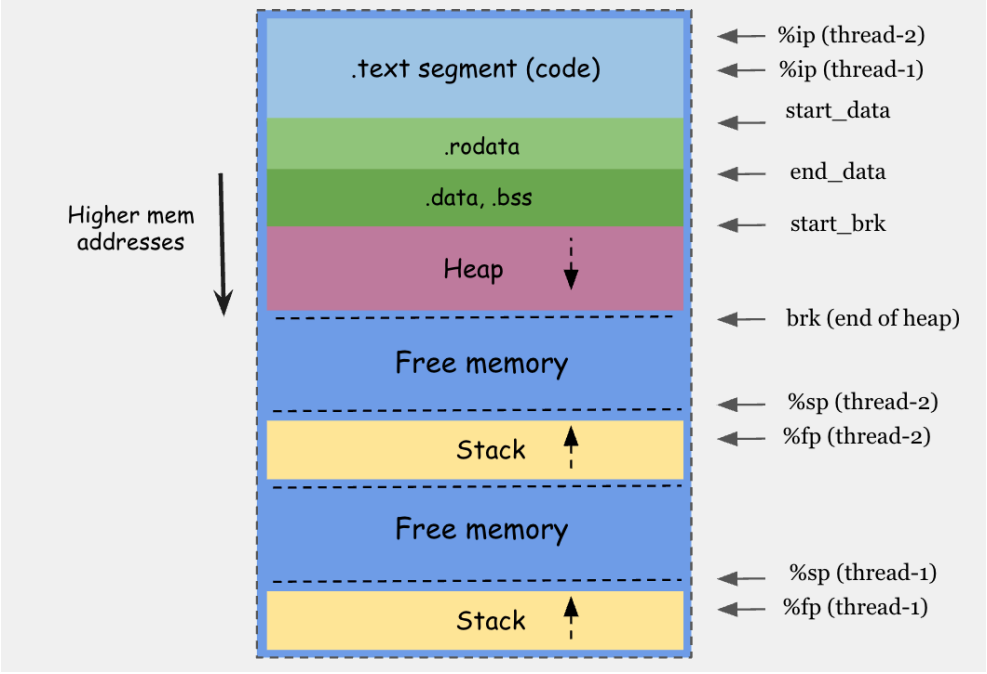
- Q1: What is a thread?

- Rethinking the use of processes: the process is a "heavy-weight" abstraction
  - Creating a new process is "expensive" because the OS must create many data structures in the PCB to manage process-specific instances of shared system resources (such as memory and open files) and keep lots of accounting metadata.
  - Communication between processes is "expensive" because it requires OS intervention since each process is a unit of fault isolation and confinement.
  - Continuously switching execution between processes has hardware side effects.
    - Conceptually, it is as simple as periodically putting the process that the processor is currently executing back at the processor's ready queue, and selecting a new process from the front of the queue.
    - In practice, because process switching involves address space switching, this leads to TLB misses.
      - On a TLB miss, the processor needs to walk the translation table present in physical memory and update its address translation entries.
      - A virtual-to-physical address translation on a TLB hit (~95% rate) is ~1 processor cycle (~0.5ns) vs. a virtual-to-physical address translation on a TLB miss, which is ~135 processor cycles.
      - The virtual-to-physical translation table is usually hierarchical, meaning there needs to be multiple memory accesses, and not just one.
      - If the translation table is not present in the physical memory then "we are in the zone" (50 μs for a disk access vs. 0.5 ns on a TLB hit).
    - There are software and hardware optimizations to alleviate the overhead of some of the above, but why the hassle? Let's take a step back.
- **What are we trying to do**? Complete as many jobs as possible in the unit of time.
  - In order to complete a job, we may need to execute lots of different tasks.
  - Recall the example of "baking a cake": Each different "mess" you make around your kitchen while following the instructions on how to bake a cake (e.g., while beating the eggs, or while beating the butter) is a different task.
  - The faster you complete all tasks, the faster the desired job is completed.
  - Since tasks require coordination (e.g., when you finish beating butter, you need to do something with it) and thus communication, if tasks are mapped to processes
    - The OS will need to intervene every time the enclosing processes execute interdependent tasks.
    - Plus, the non-negligible overhead of switching between processes.
- **We need a thinner abstraction**
  - Separate the concept of a process from its execution state.
  - The **thread** abstraction captures the minimum unit of execution.

- **Q2: What are the differences and similarities between threads and processes?**
  - "A **thread** is a single flow of control within a process, with its own thread ID, scheduling priority and policy, errno value, floating-point environment, thread-specific key/value bindings, and the required system resources to support a flow of control." IEEE Std 1003.1-2008 (POSIX.1-2008), Base Definitions, Section 3.190.
  - "A **process** is an address space with one or more threads executing within that address space, and the required system resources for those threads." IEEE Std 1003.1-2008 (POSIX.1-2008), Base Definitions, Section 3.189.
  - All threads of a process share the code, data, and heap segments, as well as all shared system resources allocated by the OS to their process.
  - Each thread has its own status (e.g., ready, running, or waiting), execution state (i.e., processor registers), and a thread-specific portion of the stack.
  - Unlike processes
    - Thread creation is inexpensive because there is no need to copy/duplicate the complete address space.
    - Context switching between threads of the same process is inexpensive because there is no address space switch and the TLB remains hot.
    - Communication between threads of the same process is inexpensive because it need not be mediated by the OS. (But it comes with new responsibilities.)
  - We'll see real examples soon, showing the overhead of threads vs processes.

Overview of a multithreaded process VAS



Multithreaded process VAS on Linux

```c
#define  NUM_THREADS 3

void print_stack_pointer(int thread_id) {
  uint64_t  sp;
  asm volatile ("mov %0, sp" : "=r" (sp));
  printf("[tid: %d]; sp: 0x%lx, &sp: %p\n",
         thread_id, sp, &sp);
}

void* foo(void* arg) {
  int thread_id = *(int*) arg;
  print_stack_pointer(thread_id);
  return NULL;
}

void main() {

  int thread_ids[NUM_THREADS];
  pthread_t threads[NUM_THREADS];

  for (int i = 0; i < NUM_THREADS; i++) {
    thread_ids[i] = i + 1;
    pthread_create(&threads[i], NULL, foo,
                   &thread_ids[i]);
  }

  for (int = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
}
```

```
--- before threads creation ----
aaaab0510000-aaaab0512000 r-xp …      thread_vmas
aaaab0521000-aaaab0522000 r--p …      thread_vmas
aaaab0522000-aaaab0523000 rw-p …      thread_vmas
aaaad9392000-aaaad93b3000 rw-p …      [heap]
ffff8d080000-ffff8d208000 r-xp …      /usr/lib/libc.so.6
…
ffff8d27e000-ffff8d27f000 r-xp …      [vdso]
ffff8d27f000-ffff8d281000 r--p …      /usr/lib/ld-linux-aarch64.so.1
ffff8d281000-ffff8d283000 rw-p …      /usr/lib/ld-linux-aarch64.so.1
ffffd83b9000-ffffd83da000 rw-p …      [stack]

[tid: 0]; sp: 0xffffd83b91a0, &sp: 0xffffd83b91b0

--- after threads creation ----
aaaab0510000-aaaab0512000 r-xp …   thread_vmas
aaaab0521000-aaaab0522000 r--p …   thread_vmas
aaaab0522000-aaaab0523000 rw-p …   thread_vmas
aaaad9392000-aaaad93b3000 rw-p …   [heap]
ffff8c060000-ffff8c070000 ---p ……
ffff8c070000-ffff8c870000 rw-p …
ffff8c870000-ffff8c880000 ---p …
ffff8c880000-ffff8d080000 rw-p …
ffff8d080000-ffff8d208000 r-xp …   /usr/lib/libc.so.6
…
ffff8d27e000-ffff8d27f000 r-xp …   [vdso]
ffff8d27f000-ffff8d281000 r--p …   /usr/lib/ld-linux-aarch64.so.1
ffff8d281000-ffff8d283000 rw-p …   /usr/lib/ld-linux-aarch64.so.1
ffffd83b9000-ffffd83da000 rw-p …   [stack]

[tid: 2]; sp: 0xffff8c86e810, &sp: 0xffff8c86e830
[tid: 1]; sp: 0xffff8d07e810, &sp: 0xffff8d07e830
```

- **Q3: How does the OS implement threads?**
  - POSIX-compliant OSes need to manage both threads and processes.
  - Conceptually, it is easy to add kernel support for the thread abstraction, since, in principle, it is a subset of the process abstraction.
    - Scheduling decisions: On threads.
    - Address space decisions: On processes.
    - Book-keeping decisions: On processes (modulo execution state).

```
do {
  Get a process P from ready queue
    Execute P until time Q expires
      Put  P back in ready queue
} while(1)
```



```
do {
    Get a thread T from ready queue
    Change address space, if needed
    Execute T until time Q expires
      Put  T back in ready queue
} while(1)
```

  - POSIX does not dictate whether threads should have their own schedulable context and run simultaneously on different processors. Therefore, threads can be implemented as
    - **User-level threads** (e.g., early Solaris "green" threads)
      - Multiple threads are mapped to one schedulable kernel context.
      - Faster to create (syscalls take ~70 cycles; function calls take ~5 cycles).
      - Invisible to the OS scheduling decisions: One thread blocks → all threads of the process block, even those with tasks ready to be executed.
      - Only one syscall per time. (One kernel stack per process.)

    - **Kernel-level threads** (e.g., glibc pthreads in Linux)
        - Each thread is mapped to one schedulable kernel context (1:1).
        - See glibc's pthread_create implementation, using the clone3 syscall.
        - Slower to create, and interact with; but, integrated with OS scheduling decisions: One thread blocks → the OS will schedule another.
    - Note that the above distinction between user- and kernel-level threads refers to the implementation details of threads, and not to whether they are instantiated by the OS kernel (e.g., via kthread_create() "kernel thread") and run exclusively in kernel's address space or by user-space programs (e.g., via pthread_create()) and have a user space portion.
- In practice, all modern OSes support kernel-level threads and let applications or language-specific thread libraries implement one of the three alternative ways for mapping their respective thread abstraction to kernel-level threads.
    - **One-to-one (1:1)**: Each user-space thread is mapped to one kernel-level thread.
    - **Many-to-one (M:1)**: Many user-space threads are mapped to a kernel-level thread.
    - **Many-to-many (M:N)**: Many user-space threads are multiplexed on top of many kernel-level threads. (E.g., go routines.)
- See this for more advanced models: https://homes.cs.washington.edu/~tom/pubs/sched_act.pdf.

- Q4: What is the historical evolution from uniprogramming to multiprogramming?

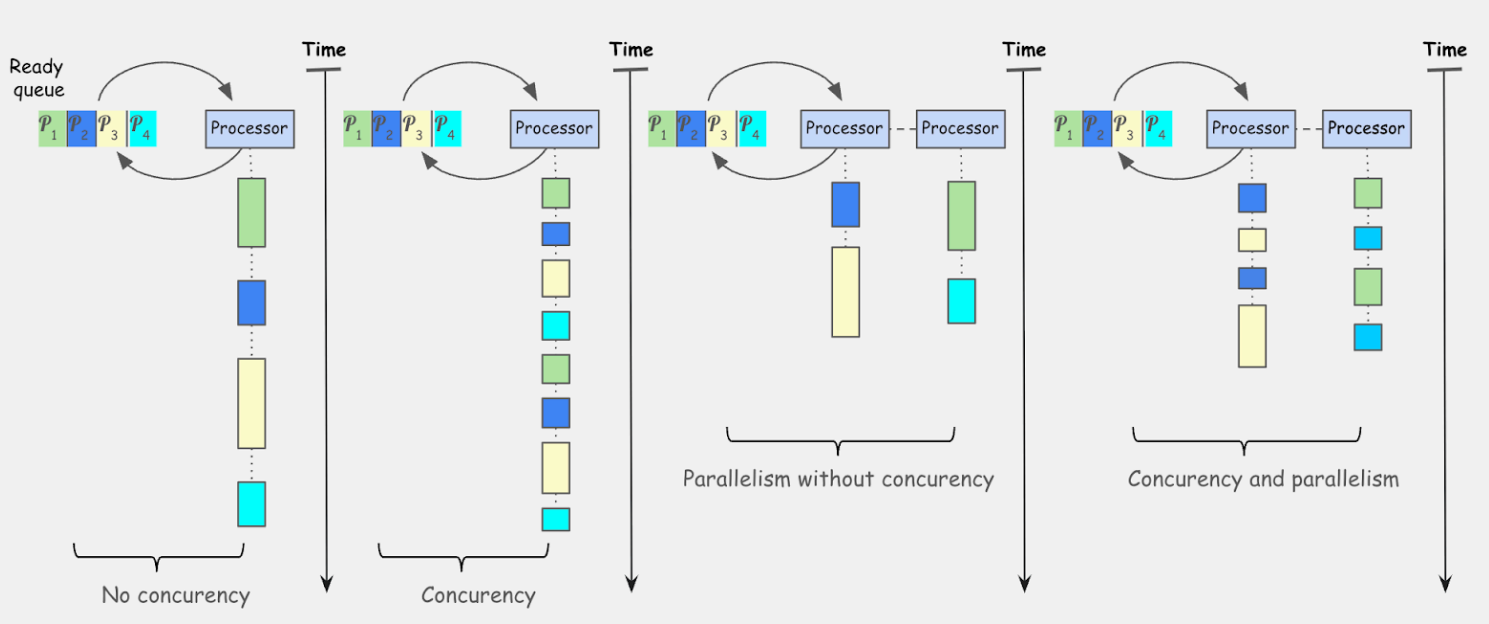
- **Uniprogramming**: Load one program in memory and execute it to completion.
    - A human operator acts as the dispatcher.
        - For each job: {load a program in memory, execute to completion}.
    - Just needs a simple library of device drivers to use primitive hardware resources.
    - OK idea for the 70's mainframes.
- **Multiprogramming**: Multiple processes reside in memory at the same time and the OS allocates the processor to each of them by switching to a new process every time the current process needs to block (e.g., while waiting for an asynchronous event).
    - There is no strict time allocation, meaning a process may keep running until it blocks, or to completion, or indefinitely.
    - OS functionality required
        - Virtual memory (for fault isolation and to keep many processes in memory).
        - Hardware interrupts (for asynchronous events; e.g., to support disk DMAs).
    - Improves processor utilization and job throughput.
    - Violates one of the three OS desirable properties. [Q: Which one?]
- **Multitasking:** Multiple processes reside in memory at the same time, and the OS allocates the processor to each of them with an upper bound on how much time each can use the processor for (regardless of whether it will even block or not).
    - Usually implemented with preemptive scheduling (e.g., **timesharing**)
        - Each processor has a dedicated timer which expires periodically.
        - On each time tick (hardware interrupt), the OS takes control and inspects bookkeeping statistics regarding how long each schedulable context has used the processor for, since the previous tick.
        - Given the cumulative scheduling statistics and the scheduling policy the OS implements (e.g., round-robin), the OS decides which schedulable context to allocate the processor to next.

- Fast switching between schedulable contexts gives the illusion that each context has a dedicated processor for itself: For example, in Linux the timer fires up 250 times / second by default, and invokes the scheduler tick function.
- OS functionality required
  - Virtual memory (for fault isolation and to keep many processes in memory).
  - Hardware interrupts (for asynchronous events; e.g., DMA and timers).
- Immediate feedback to users (i.e., each process or thread will take some processor time within milliseconds) makes the system "pleasant" to use.

- **Parallelism**: Multiple different tasks run on multiple available processors in the system.
  - Parallelism on multiprocessor systems combined with multitasking per processor is "dreamland": High throughput (resource utilization) / Low latency (interactivity).
- Now the responsibility is shifted from the OS to the programmer.

- Q5: What is concurrency?

- Concurrency is an execution paradigm where multiple tasks (i.e., execution contexts, such as processes or threads) run seemingly simultaneously on the same shared hardware resources.



- Most modern operating systems support concurrency through time–sharing–based preemptive scheduling: the OS interleaves the execution of all ready tasks, such that each runs for no more than a predefined small time quantum, and when that quantum expires, the running task is involuntarily preempted by the OS.
- With a sufficiently small time quantum—typically on the order of 10–100 ms—if a single processor handles a few dozen ready tasks, each gets to execute multiple times per second, and thus they all appear responsive to their users, akin to running simultaneously. (If multiple processors are available, the concurrent operations will, in fact, execute truly simultaneously.)

- Concurrency is a desirable OS property because it provides
    - **Responsiveness**: No task stays blocked for perceptibly long, giving users the illusion that their concurrent tasks have exclusive continuous access to all hardware resources.
    - **Throughput**: No single long-running task can monopolize the processor, allowing, overall, more tasks to execute (and potentially complete) in a unit of time.
    - **Scalability**: The more hardware resources that are available, the more concurrent tasks the system can execute on a unit of time.
- The OS has done its part to provide abstractions for multitasking and parallelization.
- Now, it's our turn to design code that takes advantage of what is being offered.
    - Threads are the most popular abstraction for concurrency.
    - Split your code into small routines expressing independent tasks.
    - Divide and conquer as much as possible and avoid algorithms with shared state.
        - If the tasks expressed in your code are independent, the OS will transparently take care of scheduling, and you only need to select the appropriate abstraction.
        - If the tasks expressed in your code are interdependent and access shared resources, then coordination is required, and the burden is on the programmer.
- **Benchmarking concurrency threads vs. processes**
    - "Given a set of N random numbers from 1 to 10 million, how many of them are primes?"
    - The choice of abstraction leads to a very different user experience
        - Processes have a scalability issue: More tasks ⇒ Performance degradation
        - Adding more hardware resources will not help alleviate the problem.
        - What this graph shows is that the processor has to execute more instructions for the OS just to instantiate (and keep track of) a process than the instructions required for solving the actual task assigned to each process.
        - Since an equally difficult task is assigned to every process (numbers are randomly chosen), the overhead degradation over time is due to the accumulation of instructions for the creation and bookkeeping of processes.

