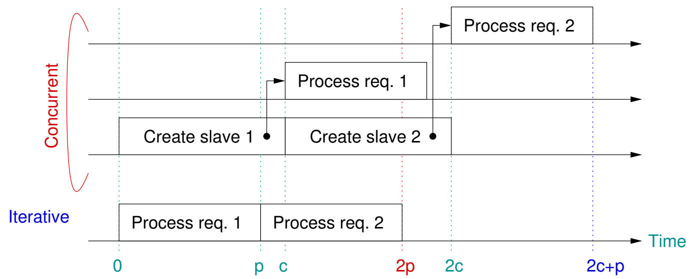
Managing concurrency

Stefan D. Bruda

CS 464/564, Fall 2023

Concurrent vs iterative, take 2

- What we did up to this point: when we needed a new thread of control, we just created a new thread or process = demand-driven concurrency
  - It may look like the right (i.e. optimal) thing to do, but this is not always the case
  - Sometimes, we are better off if we use an iterative server (sic!)
- When is iterative better? When the responses to queries are processed very quickly
  - In this case, concurrency just adds overhead



Managing concurrency

- In the general case though, concurrency does yield better performance
- We use as concurrency measure the number n of simultaneous threads that execute at a given time (be they in the same process or in different processes)
- Still, demand-driven concurrency is not necessarily the best choice
  - For one thing, n can grow unboundedly; anything that grows without bounds is bad
  - In particular, if we have tons of threads, we end up spending most of the time doing context switching (rather than useful work)
- Idea #1: limit the number of threads that can run simultaneously to a fixed limit n_{max} (how?)
  - When using threads, we can use a semaphore h
    - we initialize h with n_{max}
    - each time we create a thread we wait on h
    - each time we return from a thread we post h
  - When using processes we can simulate a semaphore by using a file holding two numbers (maximum + current) accessed within a critical region

Managing concurrency (cont'd)

- Idea #2 (variation on #1): not only we limit the number of threads that run concurrently, we also preallocate them
  - We get a bunch of threads (or processes) which do nothing at the beginning
  - If a client requests connection and we have any idle thread/process, we put it to work
  - If we do not have any idle thread/process, the incoming client will wait in the TCP queue until something becomes available
  - How do we put a thread or process in this waiting state? More precisely, how do we activate a sleeping thread/process when we need it?
    - We share the master socket (we put the call to `accept` inside the child threads)
    - A thread is idle when it blocks on the call to `accept`
    - Once a client comes, the quickest idle thread will accept the connection and this will wake it up
    - The other idle threads will continue to block on `accept`
    - The thread just woken up will then handle the client
    - Once the client finishes the interaction, the handling thread will go back to accepting new connections, and will block on `accept` in the case that no clients are asking for a connection
    - After creating the child threads, the master thread does not need to do anything else

Advantages of preallocation

- The main advantage: We reduce the system overhead, and thus we increase efficiency, response time, you name it
  - Process/thread creation does take some time, so we spend all of this time when the server starts (once a week in the middle of the night maybe) instead of spending bits of it each time a client connects
  - We practically never spend time to destroy processes or threads!
  - A good operating system will know when a thread is idle and will not select it for running on the CPU; so in a good OS we do not even do much context switching if we do not need to
- Beside reducing overhead, we also set a bound to the maximum number of threads of execution running concurrently (always a good thing)

Caveats in preallocation

- Bad news: there are caveats
- Good news: there are ways around them (basically, careful programming solves them all)
- Main problem: memory leaks
  - A memory leak happens when you allocate memory dynamically using `malloc`/`new` but fail to deallocate (all of) it when no longer needed (using `free`/`delete`)
  - Memory leaks can blow any program (system!) to pieces, but they become really critical in preallocated threads
  - Indeed, the life of preallocated threads if very long, and so they have a lot of time to accumulate leaked memory
  - In a heavy traffic server, it takes days (at best) to allocate enough virtual memory to render the system unusable
  - So be careful about memory leaks!

Caveats in preallocation (cont'd)

- Possible problem on non-Linux systems: concurrent calls to accept
  - In Linux, concurrent calls to `accept` are guaranteed to be handled properly and efficiently
  - Other Unices do not necessarily offer such guarantees
  - Concurrent calls to `accept` may not be handled at all: the first call blocks, the others return an error
  - Even when handled correctly, such handling may be awkward and inefficient
    - For instance, it may be the case that when a request arrives all the threads blocked on `accept` are unblocked
    - All but one get back into the blocked state, but meantime the CPU context switches between them a whole lot
  - Solution: avoid simultaneous blocks to `accept` by surrounding all of these system calls into critical regions
    - This guarantees that only one thread will reach the `accept` call at any given time

Delayed thread allocation

- In idea #2 we introduced a difference between the time a connection request arrives and the time when a new thread is created
  - The time difference happens to be negative (we create threads before we receive any connection request)
  - A positive time difference is just as possible (and useful)
- Idea #3 is to use delayed thread allocation
  - Remember, thread allocation consumes resources (both memory and CPU time)
  - So why create a thread when we do not need it?
  - We better begin with an iterative server (no resources consumed in allocating threads)
  - If the client is served quickly enough, we do not create anything; the (sole) server thread serves the request and only then moves to another client
  - We have thus an iterative server to begin with
  - Until the serving time passes a fixed threshold, time at which the server creates a new thread and passes to it the task at hand (of serving the current request)
    - The master thread then goes to serve another client in the same manner
  - We end up with a server that is either iterative or concurrent, the choice being dynamic (can vary from one request to the next)

The best of both worlds

- Preallocation and delayed allocation can be combined: Create new threads if necessary (using delayed allocation), but do not do this if the number of concurrent threads exceeds a given $n_{\max}$
- Of course, nothing prevents the dynamically allocated threads to be preallocated
- In practice creating threads or processes is cheap enough so that delayed thread allocation is not used very frequently