Sveučilište u Zagrebu
PMF – Matematički odsjek

Objektno programiranje (C++)

Predavanja 08 - Multithreading

Vinko Petričević

Multithreading

- Moderna računala uglavnom imaju više procesora
- Uobičajeni programi koje smo do sada vidjeli su koristili jednu nit izvršavanja programa, koja se izvodi na samo jednom procesoru (normalno kompajlirana)
- Da bismo iskoristili ostale procesore, trebamo ili kreirati novi proces (koji će biti u zasebnom adresnom prostoru) ili kreirati novu nit izvršavanja programa (thread)
- Od standarda 11, imamo zaglavlje <thread>, te u njemu klasu thread
- Kao konstruktor prima funckiju, funkcijski objekt ili lambdu, te može primati još dodatne parametre koje onda proslijeđuje toj funkciji
- Kreira novi thread i počne ga izvšavati paralelno
- Primjeri thread1, 2, 3
- Nema mehanizam vraćanja povratne vrijednosti
- Ukoliko parametre šaljemo po (neconst) referenci, to moramo naglasiti sa std::ref i moramo biti oprezni da je varijabla *živa* dokle i thread radi (to moramo paziti i ako šaljemo pointere)
- Primjeri 2, 3

Multithreading

- Funkcija join čeka da određeni thread završi sa radom.
- Funkcija detach() oslobađa vezu threada i objekta. Nakon toga se thread izvšava sve do završetka threada ili programa
- Funkcija joinable provjerava je moguće pozvati prethodne dvije funkcije
- Destruktor će srušiti program ukoliko je thread joinable! – dobro napisati wrappersku klasu – primjer 4

- Svaki thread ima i funkciju get_id() pomoću koje možemo dobiti id dretve
- U <thread> imamo i objekt this_thread koji ima funkcije get_id(), sleep_for(duration), sleep_until(time_point) i yield

- Broj procesora možemo dobiti sa thread::hardware_concurrency

Race conditions

- Ukoliko dijelimo resurse između threadova, moramo paziti da samo jedan thread može mijenjati sadržaj – primjer5
- Najjednostavniji način je koristeći mutex, ili atomic template
- #include <mutex>
- Na objektu tipa mutex imamo funkcije lock(), unlock() i try_lock()
- Postoji i wrapperska klasa std::lock_guard<std::mutex> koja ga automatski zaključava i otključava na destruktoru

- U zaglavlju atomic imamo template atomic, koja osigurava da je podatak (koji dijeli više threadova) uvećan/umanjen korektno. Nema konstruktor osim defaultnog, nego se mora eksplicitno pisati =

Signalizacija

- Ukoliko jedan thread čeka drugi thread da završi s nekim radom, postoje metode za signalizaciju kraja. Jedan loš primjer bi bio primjer6
- Dobro rješenje bi bilo koristeći condition_variable, kao u primjeru 7
- Mogli bi koristiti i future i promise kao u primjeru8. Drugi thread može i dalje nastaviti sa radom, ali glavni thread čeka dok vrijednost ne bude postavljena

Funkcije koje vraćaju vrijednost

- Future objekt možemo koristiti i za funkciju koja vraća neku vrijednost
- U zaglavlju <async> imamo funkciju async, sloja slično kao i thread prima neku funkciju (s parametrima). Ako funkcija vraća tip T, rezultat je future<T>.
- Rezultat dobivamo pozivom funkcije get()
- Za razliku od threada, async ne mora pokrenuti novi thread. Ako ne može, serijski će izvršiti funkciju na pozivu funkcije get
- get smijemo pozvati samo jednom, a imamo i funkciju valid() koja vraća da i je moguće pozvati get

- Za razliku od threada, exceptioni bačeni u async funkciji, prenose se u thread koji je pozvao funkciju (možemo ih uhvatit sa try) – primjer 9

- Možemo koristiti funkciju wait (wait_for, wait_until) koje možemo pozivati više puta, koje čekaju da završi izračun (ili istekne vrijeme)

Static

- Statičke i globalne varijable će biti dijeljene između threadova
- Ako to ne želimo, možemo varijablu navesti kao thread_local, te će svaki thread imati svoju takvu varijablu