Debugowanie od kuchni

Pisanie kodu nieodzownie jest powiązane z błędami – czy to syntaktycznymi, czy też, nieco trudniejszymi do wykrycia, błędami logicznymi. O ile ten pierwszy przypadek jest dość łatwy do znalezienia i naprawy, ponieważ pomaga nam w tym kompilator, tak już ten drugi może przyprawić o ból głowy. By nam nieco tego bólu oszczędzić, powstało wiele narzędzi oraz metodyk odpluskwiania kodu. Czy jednak kiedykolwiek zastanawialiśmy się, co tak naprawdę kryje się pod maską debuggera? Czemu on w ogóle działa – i jak działa – oraz co zrobić, gdy pracujemy w bardziej wymagających środowiskach i nie mamy dostępu do całego zaplecza narzędzi debugujących? W poniższym artykule postaram się rzucić nieco światła na powyższe zagadnienia, zaczynając od sprzętu, przechodząc przez kernel, a na przestrzeni użytkownika kończąc.

W artykule skupimy się wokół systemów operacyjnych Linux i FreeBSD i spojrzymy na debugger z ich perspektywy. Ponadto sporo czasu poświęcimy na przyjrzenie się sprzętowi. W końcu wsparcie od sprzętu wydaje się niezbędne do debugowania bardziej zawiłych problemów. Sprawdzimy zatem, co producenci architektur takich jak arm64 czy też amd64 dostarczają twórcom systemów operacyjnych oraz programów wbudowanych do ułatwienia debugowania.

DEBUGOWANIE W USERSPACE

Na rozgrzewkę przedstawię coś, co zna praktycznie każdy programista, czyli niektóre z metod debugowania aplikacji w przestrzeni użytkownika.

Zaczniemy od potężnego narzędzia, które jednak wymagać będzie pewnych nakładów finansowych. Jest to metoda tzw. gumowej kaczuszki. Jeśli ktoś nie jest wtajemniczony, to już spieszę z wyjaśnieniami: metoda ta polega na tłumaczeniu kodu linijka po linijce gumowej kaczuszce. Brzmi dość absurdalnie, jednak pozwala to zaoszczędzić pieniądze (kaczuszka wymaga jedynie jednorazowej i niewielkiej inwestycji, w przeciwieństwie do pełnoetatowego programisty), a dokładne tłumaczenie pozwala znaleźć błędy logiczne w programie. Kaczuszka dla programisty może być równie istotna jak dr. Wilson dla dr. House'a przy rozwiązywaniu jego medycznych zagadek.

Następną metodą, używaną przeze mnie nader często, jest logowanie. Takie logi, pomimo oczywistego spowolnienia programu, pozwalają szybko prześledzić działanie naszego kodu i znaleźć błędy.

Listing 1. Logowanie używane do debugowania programu

```
#ifdef DEBUG
#define DEBUG_LOG(fmt, ...) fprintf(stdout, fmt, __VA_ARGS__)
#else
#define DEBUG_LOG(fmt, ...)
#endif

static int small_array[5] = { 0, 1, 2, 3, 4 };

int access_array(int i) {
 DEBUG_LOG("Accessing element %d of the small_array\n", i);
 int val = small_array[i];
 DEBUG_LOG("Value is: %d\n", val);

 return val;
}
```

W Listingu 1 przedstawiono sposób, w jaki takie logowanie można zaimplementować już na etapie projektowania programu. Potem wystarczy ustawić flagę `DEBUG` i nasz program zaczyna być bardzo gadatliwy. Jednak nie oszukujmy się. Bardzo często dodatkowe logi dodaje się w biegu i niech pierwszy rzuci kamieniem ten, kto nigdy nie używał oryginalnych komunikatów w na szybko skleconych wywołaniach `printf`, gdy chciał zbadać, do którego miejsca jego program w ogóle dochodzi przed crashem oraz jaki stan przyjmują jego zmienne.

Wysyłanie komunikatów przez aplikację dostarcza wiele informacji, jednak każda róża ma kolce i nie inaczej jest w tym przypadku. Logowanie może być bardzo kosztowne. Przy dość prostych i łatwo reprodukowalnych błędach nie musimy się za bardzo tym przejmować. Jednak czasami błąd występuje jeden raz na milion albo w bardzo specyficznych warunkach – na przykład z powodu wyścigu między wątkami.

Spójrzmy na Listing 1. Tablica tam ma tylko pięć elementów. Sprawdzenie każdego z nich nie nastręczy żadnych problemów. Jednak co w sytuacji, gdy będzie miała ich setki tysięcy i funkcja `access_array()` zostanie wywołana dla każdego z nich? Jeśli chcielibyśmy w takiej sytuacji przeanalizować wszystkie logi, to może lepiej byłoby skorzystać z pomocy naszej kaczuszki.

Kolejnym kolcem, który może nas zranić, jest czas potrzebny do reprodukcji – zanim dostaniemy wszystkie logi w bardzo złożonym programie, może minąć mnóstwo czasu. Jeśli program wykonuje się tydzień, to logowanie każdej detalicznej informacji może go spowolnić nawet kilkusetkrotnie, w zależności od liczby informacji. Nie wspominając już o miejscu potrzebnym na dysku lub w pamięci, jeśli chcielibyśmy przechowywać je gdzieś do późniejszego użytku.

Jest jeszcze jeden istotny problem z logowaniem, w szczególności podczas programowania współbieżnego i występowaniu wszelkiego rodzaju wyścigów. Tutaj wprowadzenie opóźnienia może skutkować tym, że nie będzie możliwości zreprodukowania wyścigu.

Następnym problemem może być zmiana układu stosu lub sterty podczas wywoływania funkcji logujących. Funkcje takie jak `printf` mogą powodować alokacje, co, w nieco zaskakujący dla użytkownika sposób, potrafi zapobiec pojawianiu się crasha programu.