User-Mode Linux

Daniel Topolski Michał Kłosinski Jakub Wojtaszczyk

20 grudnia 2004

1 Wprowadzenie

1.1 Co to jest UML?

Najprościej mówiąc, User Mode Linux to program użytkownika, który udaje, że jest jądrem. Bardziej fachowo — UML to port Linuksa na Linuksa. Jest to program działający w przestrzeni adresowej i z uprawnieniami użytkownika, który symuluje działanie jądra, w szczególności pełni rolę jądra dla procesów potomnych. Korzysta do tego z funkcji systemowych zamiast interfejsu hardware'u, implementuje własne zarządzanie pamięcią wirtualną, własnego Gchedulera, etc.

User Mode Linux zaczął powstawać w 1999 roku jako łatka do jądra 2.0, zaś pierwsza wersja została opublikowana na wiosnę 2000. Jego autorem jest Jeff Dike. Od tego czasu swoją cegiełkę dołożyło wiele osób, spośród których warto wymienić Bill'a Stearns'a.

1.2 Po co to jest?

Posiadanie wirtualnej maszyny (jednej lub wielu) uruchomionej na naszym systemie operacyjnym daje wiele możliwości. Wśród nich warto wymienić:

- Testowanie nieznanych nam dystrybucji / jąder, których nie chcielibyśmy odpalać na naszej macierzystej maszynie, bo boimy się, że coś zepsują. Takie rzeczy można zupełnie bezpiecznie odpalać na UML-u, który jako proces na komputerze macierzystym nie ma uprawnień root'a, a tak naprawdę nie powinien z komputerem macierzystym mieć żadnego kontaktu. Zatem możemy pobawić się nowym bajerem i zobaczyć jak on działa bez ryzykowania życiem i zdrowiem naszego systemu.

- Eksperymenty z administrowaniem systemem. Czy kiedyś zastanawiałeś/aś się, co się stanie, gdy wykasujemy cały katalog /etc? Nie jest to eksperyment, który chcielibyśmy przeprowadzić na naszej maszynie - natomiast na UML-u możemy, najwyżej (zapewne) padnie.

- Eksperymenty z systemem. UML może symulować hardware, którego fizycznie nie mamy. Możemy zobaczyć, jak nasze oprogramowanie działa w warunkach bardzo małej lub bardzo dużej pamięci, przy ograniczonej przestrzeni dyskowej, a niedługo również przy wielu procesorach (tak przynajmniej zapowiadali twórcy UML-a).

- Test awarii systemu. Możemy doprowadzić naszego UML-a do jakiegoś niebezpiecznego stanu i próbować go z niego odratować. Przydaje się to jako doświadczenie, a takźźe, gdy mamy dwa komputery, z których jeden szwankuje, i boimy się wykonywać na nim nerwowych ruchów. Możemy na drugim postawić UML-a, postarać się doprowadzić go do podobnego stanu i sprawdzać, jak działają poszczególne środki zaradcze. Na stronie UML-a

  (`http://user-mode-linux.sourceforge.net`)

  znajdują się tzw. "disaster of the month", czyli przykładowy zepsuty UML, którego można naprawiać.

- Debugging jądra. Możemy odpalić UML-a, zmienić coś w jądrze, a następnie z zewnątrz debugować i szukać miejsca, gdzie się powiesiło. W szczególności pad jądra grozi nam tylko restartem UML-a, a nie rebootem komputera.

- Tzw. *honeypot*, czyli system podłączony do sieci z celowo wadliwymi zabezpieczeniami. Ma on na celu zwabić osoby, które chciałyby się gdzieś włamać. Dobry honeypot dookoła siebie posiada rozbudowane aplikacje do badania ruchu sieciowego, które umożliwiają zbadanie, skąd, kto i jak się włamywał. Dzięki temu możemy patrzeć, jakie techniki stosują włamywacze oraz potencjalnie namierzyć ich — a dzięki zastosowaniu UML-a nie musimy poświęcać w tym celu osobnego komputera, możemy zupełnie bezpiecznie dać im się włamać do UML-a.

- Tzw. *sandbox*, czyli piaskownica, w której można uruchomić nieznaną nam aplikację, która nie wiemy jak działa (np. podejrzewamy, że może być zawirusowana). Najgorsze, co nam grozi, to zawieszenie się UML-a (czyli z punktu widzenia systemu macierzystego - konieczność zabicia jednej aplikacji).

- *I przede wszystkim* do zrobienia zadania czwartego na SO.

1.3 Jak to działa?

Pierwszy UML działał w trybie TT, czyli Tracing Thread. Wyglądało to następująco:

- Każdy proces UML-a miał przyporządkowany sobie proces systemu macierzystego.

- Był stworzony specjalny wątek, zwany *tracing thread* — śledzący wątek (stąd nazwa TT), który wykrywał wywołania systemowe procesów UML-a.

- Wątek śledzący anulował wywołania funkcji systemowych i zmuszał proces do wejścia w jądro UML-a, które było odwzorowane w górną część przestrzeni adresowej każdego procesu.

Przechwytywanie sygnałów jest realizowane przez ptrace, który jest udostępnianą przez jądro możliwością przechwytywania przez jeden proces sygnałów innego procesu (używany np. przy debuggerach takich jak gdb). W momencie, gdy jest tworzony nowy proces UML-a, wątek śledzący wywołuje na nim PTRACE-SYSCALL i czeka. Gdy proces UML-a wywołuje funkcje systemową, jądro hosta przekazuje sterowanie wątkowi śledzącemu, który to wołanie funkcji systemowej neutralizuje, a następnie przekazuje sterowanie na powrót do wątku procesu UML-a, zmuszając go jednocześnie do wejścia w kod jądra UML-a. W tej chwili wątek śledzący przestaje przechwytywać wywołania systemowe z tego procesu (dzięki czemu będzie on mógł, znajdując się w UML-owym trybie jądra, wykonać wywołanie funkcji systemowej i zrealizować żądanie procesu użytkownika). Gdy proces zakończy pracę w trybie jądra, sygnalizuje wątkowi śledzącemu, że należy spowrotem przechwytywać wywołania systemowe.

Wady tego rozwiązania:

- Działa dość wolno, bo jest narzut kosztu na przechwytywanie wywołań systemowych i na wątki śledzące oraz na zmiany kontekstu.
- Wśród procesów systemu macierzystego znajduje się dużo procesów — po jednym na każdy proces UML-a, co powoduje mniejszą czytelność tego, co się dzieje.
- Proces widzi w swojej pamięci, że jest w niej jądro, czyli może się do niego dostać — zatem można odróżnić z wewnątrz, czy działam w UML-u, czy w normalnym systemie.
- Proces może przypadkiem zacząć pisać po jądrze, które znajduje się w jego pamięci — ten problem jest można rozwiązać przez tryb *jail*, który do ochrony jądra używa mechanizmów ochrony pamięci. To jednak powoduje olbrzymi narzut kosztu działania procesu.

Drugim rozwiązaniem jest tryb SKAS, czyli Separate Kernel Address Space. Pomysł jest taki, żeby istniał jeden proces dla wszystkich procesów użytkownika i drugi, osobny, dla jądra. Jądro zajmuje się śledzeniem procesu użytkownika i patrzeniem, czy nie wykonuje on wywołań funkcji systemowych, a następnie obsługiwaniem tych wywołań. Wymaga to jednak dodatkowych możliwości dla ptrace, w szczególności możliwości modyfikowania i czytania z pamięci procesu potomnego i klonowania procesu potomnego. Takie możliwości udostępnia łatka skas3, którą należy nałożyć na jądro hosta — jest ona dostępna na stronie

`http://user-mode-linux.sourceforge.net/dl-sf.html#Hostpatches`.

To rozwiązanie eliminuje powyższe cztery problemy. Wymaga jednak zmian w jądrze systemu macierzystego, oraz utrudnia debugging (jako, że zamiast śledzić konkretny proces, który oprócz tego, że jest śledzony przez wątek śledzący działa zupełnie normalnie trzeba zrozumieć które fragmenty procesu użytkownika odpowiadają temu procesowi, który chcemy śledzić).

2 Instalacja UML'a

2.1 Gdzie to można znaleć ?

Instalacje UML'a zaczynamy od wyboru: skompilowany plik czy źródła? Ponieważ jednym z tematów, który bedzie nas interesował jest debugg'owanie jądra, wybór musi paść na źródła. Będą nam potrzebne:

- Źródła normalnego jądra Linuksa - w naszym przypadku będzie to wersja 2.4.26. Można je ściągnąć z

  `http://ww.kernel.org`, czy `ftp://sunsite.icm.edu.pl`

- Patch na jądro - u nas będzie to wersja 2.4.26-3. Dostępny ze strony projektu:

  `http://user-mode-linux.sourceforge.net`

- Obraz systemu plików - użyjemy obrazu z systemem Slackware. Dostępny ze strony projektu:

  `http://user-mode-linux.sourceforge.net`

- UML Utilities - zestaw narzędzi. Dostępny ze strony projektu:
  `http://user-mode-linux.sourceforge.net`

Dla niektórych dystrybucji gotowe są też pakiety, które odrabiają część pracy za nas. Przykładem są Debian:
`http://packages.debian.com`, czy Gentoo:
`http://gentoo-portage.com` (lub po prostu *emerge usermode-sources* :).

2.2 Kompilacja jądra

Jądro UML'a jest oparte, a przez to i podobne, do normalnego jądra. Stąd też kompilacja odbywa się bardzo podobnie do standardowej.
Na początek stworzymy katalog i rozpakujemy źródła jądra:

```
madman@eniac madman $ mkdir uml
madman@eniac madman $ cd uml
madman@eniac uml $ tar xjf linux-2.4.26.tar.bz2
```

Teraz czas na patcha:

```
madman@eniac madman $ cd linux-2.4.26
madman@eniac linux-2.4.26 $ bzcat uml-patch-2.4.26-3.bz2 | patch -p1
```

W tej chwili gotowi jesteśmy do konfiguracji jądra. Mamy tutaj do wyboru *make config*, *make menuconfig*, *make xconfig*. Dodatkowo musimy ustawić architekturę *ARCH=um*.

```
madman@eniac linux-2.4.26 $ make menuconfig ARCH=um
```

Domyślna konfiguracja nas zazwyczaj zadowala, ale na wszelki wypadek sprawdźmy opcje które będą nam później potrzebne do debug'owania jądra.Są to:

```
Kernel hacking  --->
    [*] Enable kernel debugging symbols
    [*] Enable ptrace proxy
```

Gdy mam konfiguracje za sobą, czas na kompilację:

```
madman@eniac linux-2.4.26 $ make linux ARCH=um
```

W tym miejscu możemy już też pomyśleć o zbudowaniu modułów:

```
madman@eniac linux-2.4.26 $ make modules ARCH=um
```

Po tych czynnościach mamy już zbudowane i gotowe do działania jądro. Warto zwrócić uwagę na rozmiar pliku z jądrem(ok. 34MB). Jest on spowodowany obecnością syboli debug'owania. Jądro można odchudzić do normalnych rozmiarów poleceniem *strip*.

2.3 System plików

Tym czego nam teraz brakuje to obraz systemu plików. Zmieńmy nazwę obrazu na *root_fs*. Będzie on wtedy domyślnie ładowny przy uruchomieniu UML'a. Przy okazji umieszczamy w wygodny miejscu skompilowane jądro:

```
madman@eniac linux-2.4.26 $ cp linux ../
madman@eniac linux-2.4.26 $ cd ..
madman@eniac uml $ mv root_fs_slack_8.1 root_fs
```

2.4 Instalowanie modułów

Kolejnym krokiem będzie umieszczenie modułów na swoim miejcu. Można to zrobić na wiele sposobów. Ja pokażę dwa z nich.

2.4.1 Instalowanie modułów z poziomu systemu hosta

Cały proces instalacji modułów przeprowadzamy bez włączonego UML'a.
Na początek stworzymy katalog pod który zaczepiamy(mount) obraz systemu plików (wymaga to włączonej opcji *Loopback device support* w konfiguracji jądra systemu hosta):

```
madman@eniac uml $ mkdir mnt
madman@eniac uml $ mount root_fs mnt -o loop
```

Następnie instalujemy, wcześniej skompilowane, moduły w zmienionym katalogu:

```
madman@eniac uml $ cd linux-2.4.26
madman@eniac linux-2.4.26 $ make modules_install \
INSTALL_MOD_PATH=~/uml/mnt
```

2.4.2 Instalowanie modułów z poziomu UML'a

Uruchamiamy UML'a,tworzymy katalog i podczepiamy(mount) pod niego katalog ze źródłami UML'a z systemu hosta. Następnie instalujemy moduły:

```
madman@eniac uml $ ./linux
root@darkstar:~# mkdir src
root@darkstar:~# mount none src -t hostfs -o /home/madman/uml/linux-2.4.2
root@darkstar:~# cd src
root@darkstar:~/src# make modules_install
```

2.5 UML Utilities

Aby zapewnić poprawne funkcjonowanie naszego User Mode Linuxa, musimy zainstalować zestaw narzędzi. Instalacja przebiega standardowo(wymaga uprawnień root'a):

```
madman@eniac uml $ tar xjf uml_utilities_20040406.tar.bz2
madman@eniac uml $ cd tool
madman@eniac tool $ make
madman@eniac tool $ su
madman@eniac tool $ make install
```

3 Konfigurowanie Połączeń Szeregowych

Ta część prezentacji omawia następujące zagadnienia:

- Połączenia różnych kanałów Wejścia/Wyjścia hosta z różnymi typami portów szeregowych UML-a
- Urządzenia(Devices) i Kanały(Channels)
- Pseudoterminale
- Terminale
- Xterms
- Porty
- Deskryptory

3.1 Połączenia różnych kanałów Wejścia/Wyjścia hosta z różnymi typami portów szeregowych UML-a

Porty szeregowe UML-a podłączamy do urząadzeń pty(lub pts), tty, deskryptorów plików i portów hosta. Dzięki temu możemy:

- podłączyć konsolę User-mode Linuksa z nieużywaną konsolą hosta
- złączyć ze sobą dwie wirtualne maszyny poprzez połączenie jednej z urządzeniem pty, a drugiej z odpowiadającym mu urządzeniem tty
- udostępnić wirtualną maszynę przez sieć, dzięki podłączeniu jej konsoli do portu hosta.

3.2 Urządzenia i Kanały

Połączenia są realizowane w lini poleceń w następujący sposób: device=channel. Ustalane jest połączenie miedzy device z wirtualnej maszyny, a kanałem channel hosta.

- device - jest to con lub ssl (console or serial line), możemy przypisać szczególnej consoli (analogicznie dla ssl) kanał albo wszystkim: con4=tty:/dev/tty1 albo con=tty:/dev/tty1 .

  Podawanie numeru urządzenia powoduje nadpisanie ogólniejszej definicji, przy czym kolejność nie ma znaczenia np.:

  ssl0=tty:/dev/tty0 ssl=pty

  pty zostało przypisane do każdego z wyjątkiem drugiego portu szeregowego hosta.

- channel - pseudoterminale, terminale, xterm-s i porty hosta. Opisane kolejno w dalszej części prezentacji.

3.3 Pseudoterminale

device=pty lub device=pts, przypisujemy urządzeniu device wolny pseudoterminal.
con1=pty
Konsoli nr. 1 przypisano pseudoterminal, którego nazwa zostanie wyświetlona. Aby przekonać się jaki terminal odpowiada dołączonemu pseudoterminalowi należy edytować plik boot.log
.
Połączyć sie z terminalem możemy urzywając programu minicom:
minicom -o -p /dev/ttyxx
/dev/ttyxx - jest to plik terminala, xx - numer

3.4 Terminale

device=tty: /dev/ttyxx, przypisujemy urządzeniu device terminal odpowiadający plikowi /dev/ttyxx, np.:
con3=tty: /dev/tty2
Konsoli nr. 3 przypisalismy termianal, powinno istnieć też urządządzenie, które korzysta z pseudoterminala przyporządkowanego temu terminalowi.

3.5 Xterm

device=xterm urządzeniu maszyny wirtualnej mozemy przypisać wolny xterm

3.6 Porty

device=port: n, n jest numerem portu, np. 9000. Używając telnetu możemy skontaktować sie z urządzeniem device.
ssl=port: 9000

Dla każdej aktywnej sesji telnetu (ustawionej na port 9000) przydzilane jest inne urządzenie. Jeśli jest więcej sesji niż urządzeń UML-a to kolejne sesje są blokowane.

3.7 Deskryptory

device=fd:n, przypisujemy zainicjowny(!) deskryptor urządzeniu device.

con0=fd:0,fd:1 con=pty

Często stosowana kombinacja, wszystkie konsole oprócz głównej zostały przypisane w tym wypadku psedoterminalowi.

4 Praca w sieci

Ta część prezentacji dotyczy połączeń UML-a z siecią, omawiane zagadnienia:

- Sposoby wymiany pakietów
- Podstawowe ustawienia i demony
- Nadawanie adresów ethernet-owych
- Konfiguracja adresów po stronie UML-a
- Multicast
- TUN/TAP
- pcap
- Ręczna konfiguracja hosta

4.1 Sposoby wymiany pakietów

Kolejne slajdy opisują komunikację za pośrednictwem sieci z hostem, innymi maszynami w sieci lokalnej, itp..

Pomocnym programem przy konfiguracjii sieci jest uml_net (choć wymaga uprawnień root-a). Wirtualna maszyna może korzystać z siedmiu metod wymiany pakietów:

- ethertap

  Używany dla hosta z jądrem 2.2, dla wersji jąder 2.4 zalecany jest TUN/TAP

- TUN/TAP

  TUN/TAP jest w stanie użyć prekonfigurowanego urządzenia, co pozwala uniknąć używania uml_net. TUN/TAP podobnie jak ethertap, slip i slirp pozwalają maszynie UML na wymianę pakietów z hostem, który może działa jako router i udostępniać połączenia z innymi komputerami bądź maszynami wirtualnymi.

- Multicast

  Potrzebujesz wirtualnej sieci i nie chcesz konfigurować niczego oprucz UML-a.

- Switch

  Jak wyżej, ale jest to wydajniejsze rozwiązanie jeśli nie masz nic przeciwko uruchamianiu demona.

- slip

  slip jest alternatywą gdy masz problem z uruchomieniem ethertap-a lub TUN/TAP-a.

- slirp

  slirp daje użytkownikowi możliwość korzystania z sieci w przypadku braku uprawnień root-a oraz gdy nie chcemy nadawać adresu IP wirtualnej maszynie.

- pcap

  Ten sposób wymiany pakietów udostępnia interfejs tylko do odczytu, najczęściej używany do przechwytywania i filtrowania pakietów z sieci. Wykorzystuje się go do tworzenia monitorów ruchu sieciowego lub sniffer-ów.

4.2 Podstawowe Ustawienia i demony

1. Jeśli kompilujemy jądro UML-a sami, w menu 'Network device support' należy uaktywnić 'Network device suport' i trzy środki transportu danych.

2. Udostępniamy urządzenia sieciowe (interfejsu) wirtualnej maszynie UML-a. Robi się to, podając odpowiednie parametry przy uruchamianiu UML-a, shemat:

   eth <n> = <transport> , <transport args>

   Na przykład wirtualna karta sieciowa może zostać podłączona do ethertap-u hosta w ten sposób:

   eth0=ethertap,tap0,fe:fd:0:0:0:1,192.168.0.254

   To każe interfejsowi eth0 wewnątrz wirtualnej maszyny podłączyć się do urządzenia /dev/tap0 należącego do hosta, przypisuje mu adres ethernet-owy, i przypisuje interfejsowi tap0 hosta adres IP. Oba adresy muszą sie różnić.

3. Uruchamiamy UML-a, konfigurujemy urządzenia UML-a, ustalamy ścieżki do komunikacji. Teraz możemy nawiązać kontakt z dowolną maszyną podłączoną do sieci.

4. Używając Management Console można dodawa i usuwać urządzenia sieciowe w trakcie działania UML-a.

5. Gdy ifconfig w UML-u zgłosi błąd i sieci nie da się postawić, odpalamy dmesg i patrzrymy co się stało.

6. Zapewne będziemy potrzebować uml_helper-a lub demona switch. Oba narzędzia instalowane są z RPM'a, ale jeśli jądro kompilowaliśmy sami, musimy wyciągnąć je z CVS'a, skompilować i zainstalować (/tools/uml_net i /tools/uml_router). Kompilacja standardowa (make), instalujemy do /usr/bin, dodatkowo uml_net (uml_helper) musi mieć ustawiony suid.

4.3 Nadawanie adresów ethernet-owych

TUN/TAP, ethertap i switch pozwalają na ręczne przypisanie adresów sprzętowych wirtualnym kartom sieciowym, nie jest to jednak konieczne. Jeśli nie podamy sami adresu, sterownik wygeneruje adres i bazując na adresie IP maszyny (postaci: fe:fd:nn:nn:nn:nn, gdzie nn.nn.nn.nn to adres IP). Zazwyczaj to wystarcza, by zapewnić unikatowy adres fizyczny. Wyjątkami są sytuacje, gdy:

- Są różne zestawy wirtualnych kart sieciowych w naszej sieci i używają innego sposobu adresownia
- Nie chcemy używać urządzenia do pracy w sieci IP, więc nie ma ono numeru IP

Jeśli pozwalamy sterownikowi przypisać adres sprzętowy, musimy mieć pewność, że zna on adres IP interfejsu, zanim zostanie on włączony. W UML-u wywolujemy:

UML# ifconfig eth0 192.168.0.250

Przypisując adres sprzętowy ręcznie, należy się upewnić co do parzystości pierwszego bajtu. Adresy z nieparzystym pierwszym bajtem używane są do rozgłaszania.

4.4 Konfiguracja interfejsu po stronie UML-a

Po opisaniu urządzenia sieciowego w linii poleceń, odpalamy UML-a i logujemy się. Następnie podnosimy interfejs sieciowy:

UML# ifconfig ethn ip-address up

W tej chwili powinniśmy mieć możliwość pingowania hosta. Dostęp do pozostałej części sieci uzyskujemy ustawiając domyślną ścieżkę do hosta:

UML# route add default gw host ip

Dla host-a o adresie 192.168.0.4, wygląda to tak:

UML# route add default gw 192.168.0.4

Musimy pamiętać, by nie ustawić przypadkiem ścieżki do sieci lokalnej!

Uwaga: jeśli nie można nawiązać połączenia z innymi komputerami w sieci (tej fizycznej), może to być spowodowane przez automatycznie ustawioną ścieżkę. Jeśli po odpaleniu routen zobaczymy coś takiego:

Destination Gateway Genmask Flags Metric Ref

192.168.0.0 0.0.0.0 255.255.255.0 U 0 0

Use Iface

0 eth0

jeśli maska inna niż 255.255.255.255, zamieniamy tą ścieżkę na ścieżkę która prowadzi do hosta:

UML# route del -net 192.168.0.0 dev eth0 netmask 255.255.255.0

UML# route add -host 192.168.0.4 dev eth0

Domyślna ścieżka po stronie hosta pozwoli UML-owi na wymianę pakietów z każdą maszyną w sieci.

4.5 Multicast

Najprostszym sposobem na postawienie wirtualnej sieci jest zastosowanie mcast-a dostępnego w UML-u w wersji 2.4.5-5um i późniejszych. Nasz system musi mieć włączony multicast (można to zrobić w trakcie kompilacji jądra), a host musi mieć urządzenie, które obsługuje multicast. Zazwyczaj jest to eth0, ale jeśli nie mamy karty sieciowej, UML wywali dziwne błędy przy uruchamianiu interfejsu.

Wystarczy uruchomić dwa UML-e z eth0=mcast.W linii poleceń logujemy się i konfigurujemy interfejsy po stronie maszyn UML-owych:

UML1# ifconfig eth0 192.168.0.254

UML2# ifconfig eth0 192.168.0.253

Pełny zestaw opcji z linii poleceń dla mcasta to:

ethn=mcast,ethernet address,multicast address,multicast port,ttl

4.6 TUN/TAP

Najprostrzym sposobem uruchomienia jest odpalenie uml_net i pozwolenie mu na skonfigurowanie hosta. Ta metoda jest zalecana dla początkujących. Jeśli podamy adres IP dla urządzenia po stronie hosta, uml_net wykona za nas resztę - jedynym wymaganiem jest dostępność TUN/TAP-a.

To jest format opcji w lini poleceń służącej do podłączenia UML-a do TUN/TAP-a:

eth <n> =tuntap,,, <host IP address>

Np. to pozwoli podpiąć eth0 UML-a do pierwszego dostępnego urządzenia tap (po stronie hosta) i przypisanie mu (tap-owi) adresu IP.

eth0=tuntap,,,192.168.0.254

Gdy jak w tym przypadku używamy uml_net-a, musimy być świadomi, że zmiana numeru IP po stronie UML-a zostanie zauważona przez uml_net i zostaną wprowadzone odpowiednie zmiany do tablic routingu i arp hosta. To jeden z powodów, dla których nie należy używać uml_net, jeśli zachodzi podejrzenie, że jeden z użytkowników może być nieprzyjazny. Jeśli np. UML podszyje się pod serwer DNS, to host może zacząć kraść pakiety przeznaczone dla serwerów DNS w sieci i dostarczać je do tego UML-a.

Z używaniem TUN/TAP-a na jądrze 2.4 wiążą się następujące problemy:

- TUN/TAP nie działa z jądrem 2.4.3 i wcześniejszymi.
- Na zaktualizowanym jądrze TUN/TAP może zakończyć prace z błędem:

File descriptor in bad state

Rozwązeniem jest zmiana dowiązania /usr/src/linux tak, by wskazywało na pliki nagłówkowe działającego jądra.

4.7 pcap

Pcap podpinany jest do karty sieciowej UML-a przez parametr w linii poleceń:
ethn=pcap,host interface,filter expression,option1,option2
Przy czym "filter expression" i opcje nie są wymagane, przykład:
eth0=pcap,eth0,tcp
eth1=pcap,eth0,!tcp
Interfejs eth0 UML-a zacznie emitować pakiety tcp z interfejsu eth0 hosta, a eth1 UML-a wszystkie pozostałe pakiety z tego interfejsu.

4.8 Ręczna konfiguracja hosta

Gdy brak adresu dla urządzenia ethertap lub slip po stronie hosta, uml_net nie dokona zmian w jego konfiguracji i będziemy zmuszeni zrobić to sami. W tym przykładzie 192.168.0.251 to adres po stronie hosta, a 192.168.0.250 to adres po stronie UML-a.

1. Najpierw podnosimy tap i slip

```
host\#  ifconfig tap0 arp mtu 1484 192.168.0.251 up

host\#  ifconfig sl0 192.168.0.251 pointopoint 192.168.0.250 up
```

2. Teraz modyfikujemy tablice routingu w UML-u

```
UML\#  route add -host 192.168.0.250 gw 192.168.0.251
```

3. Pokazujemy sieć na zewnątrz (musimy postawić proxy arp)

```
host\#  arp -Ds 192.168.0.250 eth0 pub
```

4. Jeszcze routing na hoście

```
host\#  echo 1 > /proc/sys/net/ipv4/ip_forward
```

5 Systemy plików

UML jako systemy plików wykorzystuje pojedyncze pliki swojego hosta, zainstalowane jako urządzenia /dev/ubdX. Aby podłączyć taki plik, jak było powiedziane w dziale instalaćja, należy dodać linię

```
... ubd3=nasz_plik
```

jako opcję przy uruchamianiu UML-a.

5.1 Mechanizm COW — Copy On Write

Jeżeli z jednego systemu plików będziemy ładować wiele UML-i (np. jest to obraz systemu plików zawierającego pliki testowanej przez nas dystybucji), to warto użyc mechanizmu współdzielenia tego obrazu. Próba ręcznego odpalenia wszystkich UML-i z tego systemu przez wpisanie im

```
... ubd0=bazowy_system_plikow
```

do linii komend skończy się tak, jak próba odpalenia więcej niż jednego systemu operacyjnego z jednego dysku logicznego, czyli uszkodzeniem systemu plików. Naley skorzystać z mechanizmu COW - czyli plik bazowy uczynić plikiem tylko do odczytu (poprzez

```
chmod 444 system
```

) a następnie każdego UML-a zmusić, by trzymał u siebie informację o tym, jakich modyfikacji dokonał. W ten sposób zaoszczędzimy wiele miejsca na dysku do czasu, kiedy UML-e zaczną dokonywać w systemach plików dużych zmian. Robimy to poprzez dodanie do linii komend uruchamiających UML-a

```
... ubd0=nasz_plik.cow,nasz_plik
```

gdzie pierwszy plik zostanie utworzony jako plik COW, a drugi będzie plikiem tylko do odczytu zawierającym dane bazowe. Jeżeli UML dokonał dużych zmian na pliku bazowym, i chcemy, aby dalej pracował na własny koszt, to używamy do tego polecenia

```
host% uml_moo nasz_plik,cow,nowy_plik_podstawowy
```

Oczywiście po wykonaniu tej operacji ta kopia UML-a nie będzie już działała w trybie COW.

5.2 Tworzenie systemu plików

Tworzenie systemu plików dla UML-a jest proste. Wystarczy stworzyć pusty plik (najlepiej rzadki, żeby nie zajmować przestrzeni dyskowej - przykładowo poleceniem dd). Ten plik podłączamy jako urządzenie /dev/ubdX w linii komend do uruchamianego UML-a. Ten będzie go widział jako czysty kawałek dysku, trzeba więc założyć na nim system plików, którego będziemy chcieli używać (np. poprzez mkreiserfs). I już. Konkretnie, piszemy np. tak:

```
host% dd if=/dev/zero of=new_filesystem seek=100 count=1 bs=1M
host% ./linux ... ubd5=new_filesystem
host# mkreiserfs /dev/ubd/4
UML# mount /dev/ubd/4 mnt
```

5.3 Rozszerzanie systemów plików

Aby zwiększyć jakiś system plików wychodzimy z UML-a, sprawdzamy spójność systemu plików (na wszelki wypadek) przez e2fsck, lub inny odpowiedni do naszego systemu plików. Nastepnie przez dd z opcją conv=notrunc dodajemy na koncu odpowiednią ilość pustej przestrzeni (przez opcję seek możemy ją dodać logicznie, bez zaśmiecania dysku zerami), a następnie używamy resize2fs lub innego odpowiedniego dla naszego systemu plików programu i (po ewentualnym ponownym e2fsck na wszelki wypadek) odpalamy UML-a z nowego systemu plików. Przykładowo:

```
host% e2fsck -f plik_systemu_plikow_UMLa
host% dd if=/dev/zero of=plik_systemu_plikow_UMLa bs=1 count=1
seek=nowy_rozmiar conv=notrunc
host% resize2fs -p plik_systemu_plikow_UMLa
host% e2fsck -f plik_systemu_plikow_UMLa
```

Oczywiscie wszystkie powyzsze operacje trzeba wykonywać urpzednio wyłączywszy UML-a i odpalić go na nowo dopiero po zakończeniu całej operacji.

5.4 Dostęp do plików hosta

Oczywiście do hosta można dostać sie przez sieć. Ale przecież UML jako proces użytkownika na hoście powinien móc bezpośrednio dostać się do potrzebnych mu plików. Najprostszym sposobem na to jest zamontowanie przez

```
UML# mount -t hostfs none /mnt/host
```

systemu hosta na naszym systemie. To powoduje zainstalowanie katalogu

```
/
```

na /mnt/host naszej maszyny wirtualnej. Jeżeli nie chcemy udostępniać całego systemu plików hosta, a tylko jego fragment, dodajemy opcję -o, np:

```
UML# mount -t hostfs none /mnt/host -o /home
```

Można też skopiować odpowiedni fragment obrazu systemu plików hosta na obraz systemu plików:

```
host# mount -o loop moj_root_fs /mnt/new.root
host# mount -o loop source_fs /mnt/old.fs
host# cp -fra /mnt/old.fs /mnt/new.root
host# umount old.fs
host# umount new.root
```

6 Debug'owanie jądra

Opisana tu procedura debug'owania dotyczy UML'a pracującego w trybie trace thread. Praca w trybie skat zostaje, pominięta gdyż nie dotyczy zadania laboratoryjnego. Zainteresowanym polecam stronę projektu UML.

Jądro UML'a uruchamia się w systemie jako zwyczajny proces, a jako taki może być debug'owane. Peną różnicą jest to, że wątki jądra, są już śledzone przez ptrace'a w celu przechwytywania wywołań systemowych. Stąd gdb, nie może wykorzystać mechanizmu ptrace'a. Został jednak stworzony mechanizm omijający ten problem.

6.1 Uruchamianie jądra pod gdb

Uruchomienie UML'a z opcją *debug* powoduje rozpoczęcie procesu debug'owania od samego startu jądra. Zostaje otwarty terminal, w którym uruchamiane jest gdb. Wykonywanie jądra zstaje zatrzymane na *start_kernel*. Z tego punktu można zacząć prace z gdb.

Innym sposobem rozpoczęcia pracy z gdb jest wysłanie sygnału USR1 do procesu śledzącego wątki(trace thread). Numer pid procesu śledzącego jest wypisywany w trakcie startowania UML'a.

```
madman@eniac uml $ kill -USR1 pid_procesu
```

6.2 Podstawowe komendy gdb

Oto krótki słowniczek komend, które będą się pojawiać:

- r - run - uruchom program
- c - continue - kontynuuj wykonanie
- s - step - zrób krok
- b - breakpoint - ustawia punkt zatrzymania
- det - detach - odłacza katualnie debug'owany proces
- att - attach - podłącza się do procesu
- bt - backtrace - wyświetla zawartość czubka stosu

6.3 Debug'owanie śpiącego procesu

Nie zawsze wystarcza nam debugowanie działającego procesu. Może się zdarzyć sytuacja, w której jakiś proces np. uśnie na semaforze i się nie obudzi. Wtedy przerwanie oczekiwnaia Ĉ i wywołanie backtrace'a nic nam nie da, gdyż wyświetli się stos procesu idle. W takiej sytuacji musimy ustalić jaki proces unął gdzie nie powinien oraz ustalić jego pid - czynności proste. Następnie odłaczymy się od aktualnego procesu i podłączymy do interesującego nas procesu:

```
(gdb) det
(gdb) att pid
(gdb) bt
```

Gdy zgromadzimy inteteresujące informacje możemy wrócić do aktualnego wątku i kontynuować:

```
(gdb) det
(gdb) att 1
(gdb) c
```

6.4 Debug'owanie modułów

W gdb możliwe jest debug'owanie dynamicznie ładowanego kodu. Pozwala to na debug'owanie działania modułów. Możliwość ta nie należy jednak do łatwych do wykorzystania. Należy w tym celu wskazać plik obiektowy, który został załadowany oraz wskazać jego adres w pamięci. Dopiero wtedy gdb może odczytać symbole. Przy ponownym ładowaniu modułu trzeba skolei zadbać o skasowaniu informacj o symbolach pozostających w pamięci gdb.

Z pomocą przychodzi skrypt umlgdb z pakietu UML Utilities. Aby skrypt zadziałał należy mu ustawić pewne zmienne, mianowicie zmodyfikować listę naszych modułów. Lista ta znajduje się w treści sktyptu :

```
set MODULE_PATHS {
   "fat" "/usr/src/uml/linux-2.4.18/fs/fat/fat.o"
   "isofs" "/usr/src/uml/linux-2.4.18/fs/isofs/isofs.o"
   "minix" "/usr/src/uml/linux-2.4.18/fs/minix/minix.o"}
```

Teraz wystarczy uruchomić skrypt w katalogu, ze źródłami naszego UML'a. Skrypt mówi co dalej robić:

```
GNU gdb 5.0rh-5 Red Hat Linux 7.1
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, a
welcome to change it and/or distribute copies of it under certain
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for
This GDB was configured as "i386-redhat-linux"...
(gdb) b sys_init_module
Breakpoint 1 at 0xa0011923: file module.c, line 349.
(gdb) att 1
```

Po naciśnięciu ENTER, możemy kontynuować:

```
Attaching to program: /home/jdike/linux/2.4/um/./linux, process 1
0xa00f4221 in __kill ()
(gdb) c
Continuing.
```

Od tego momentu możemy normanie debug'owac. Gdy załadujemy moduł skrypt zareaguje autoamgicznie:

```
*** Module hostfs loaded ***
  Breakpoint 1, sys_init_module (name_user=0x805abb0 "hostfs",
      mod_user=0x8070e00) at module.c:349
  349             char *name, *n_name, *name_tmp = NULL;
  (gdb) finish
  Run till exit from #0  sys_init_module (name_user=0x805abb0 "hostfs",
      mod_user=0x8070e00) at module.c:349
  0xa00e2e23 in execute_syscall (r=0xa8140284) at syscall_kern.c:411
  411             else res = EXECUTE_SYSCALL(syscall, regs);
  Value returned is $1 = 0
  (gdb) p/x (int)module_list + module_list->size_of_struct
  $2 = 0xa9021054
  (gdb) symbol-file ./linux
  Load new symbol table from "./linux"? (y or n) y
  Reading symbols from ./linux...
  done.
  (gdb) add-symbol-file /home/jdike/linux/2.4/um/arch/um/fs/hostfs/hostfs
  add symbol table from file "/home/jdike/linux/2.4/um/arch/um/fs/hostfs/
          .text_addr = 0xa9021054
  (y or n) y
  Reading symbols from /home/jdike/linux/2.4/um/arch/um/fs/hostfs/hostfs.
  done.
(gdb) p *module_list
  $1 = {size_of_struct = 84, next = 0xa0178720, name = 0xa9022de0 "hostfs
    size = 9016, uc = {usecount = {counter = 0}, pad = 0}, flags = 1,
    nsyms = 57, ndeps = 0, syms = 0xa9023170, deps = 0x0, refs = 0x0,
    init = 0xa90221f0 <init_hostfs>, cleanup = 0xa902222c <exit_hostfs>,
    ex_table_start = 0x0, ex_table_end = 0x0, persist_start = 0x0,
    persist_end = 0x0, can_unload = 0, runsize = 0, kallsyms_start = 0x0,
    kallsyms_end = 0x0,
    archdata_start = 0x1b855 <Address 0x1b855 out of bounds>,
    archdata_end = 0xe5890000 <Address 0xe5890000 out of bounds>,
    kernel_data = 0xf689c35d <Address 0xf689c35d out of bounds>}
  >> Finished loading symbols for hostfs ...
```