دانشگاه صنعتی شریف
دانشکده مهندسی کامپیوتر

گزارش پروژه
درس سیستم‌عامل

پشتیبانی از شبکه در سطح هسته در سیستم‌عامل ۶XV

نگارش

فاطیما تیمارچی - ۴۰۲۱۰۵۸۰۲
آیه صابری - ۴۰۲۱۰۶۱۴۵
متین باقری - ۴۰۲۱۰۵۷۲۷
حورا عابدین - ۴۰۱۱۰۶۲۰۹

استاد

دکتر حسین اسدی

بهمن ماه ۱۴۰۴

فهرست مطالب

فهرست تصاویر

فصل ۱

مقدمه

در این پروژه، پشتیبانی شبکه را در سیستم‌عامل آموزشی تکمیل کردیم. تمرکز اصلی ما پیاده‌سازی درایور کارت شبکه‌ی E۱۰۰۰ در سطح kernel و اتصال آن به stack network سیستم‌عامل بود. این کارت شبکه در محیط QEMU شبیه‌سازی می‌شود، اما از دید kernel رفتاری مشابه یک کارت شبکه‌ی واقعی دارد. به همین دلیل لازم بود بخش‌های مربوط به TX و ring RX را کامل کنیم، با register های کارت شبکه که در حافظه در دسترس هستند کار کنیم و interrupt های ایجادشده را مدیریت کنیم تا ارتباط بین سخت‌افزار و هسته به‌درستی برقرار شود.

۱-۱ تعریف مسئله

مسئله‌ی اصلی پروژه ایجاد یک مسیر کامل برای ارسال و دریافت packet های شبکه داخل سیستم‌عامل بود. در بخش ارسال (TX) ، زمانی که stack network قصد ارسال یک packet را داشت، آدرس buffer داده را داخل ring TX قرار دادیم و با به‌روزرسانی register مربوطه مانند ،TDT کارت شبکه را از وجود داده‌ی جدید مطلع کردیم. پس از این‌که کارت شبکه ارسال را کامل کرد و بیت وضعیت (DD) را فعال کرد، buffer را با kfree آزاد کردیم تا مدیریت حافظه به‌درستی انجام شود.

در مسیر دریافت (RX) ، کارت شبکه پس از دریافت packet از شبکه، داده را از طریق DMA داخل buffer مشخص‌شده می‌نوشت و یک interrupt ایجاد می‌کرد. در تابع مربوط به دریافت، ring RX را بررسی کردیم، های‌packet جدید را به لایه‌ی بالاتر تحویل دادیم و سپس برای هر slot مصرف‌شده یک buffer جدید با kalloc اختصاص دادیم تا حلقه برای دریافت packet های بعدی آماده بماند.

۱-۲ اهداف پروژه

هدف این پروژه تکمیل پشتیبانی UDP در سیستم‌عامل و فراهم کردن امکان دریافت داده در space user بود. برای این کار، syscall های bind ، unbind و recv را پیاده‌سازی کردیم تا هر process بتواند روی یک port مشخص منتظر دریافت داده بماند. در لایه‌ی IP بررسی کردیم که packet دریافتی از نوع UDP باشد و port مقصد آن قبلاً bind شده باشد. در صورت برقرار بودن این شرایط، packet در queue مربوط به همان port قرار می‌گیرد. برای هر port حداکثر ۱۶ packet در queue نگه می‌داریم تا مصرف حافظه کنترل شود و اگر صف پر باشد، packet جدید drop می‌شود. همچنین در تمام مسیر دقت کردیم هر buffer که دیگر مورد نیاز نیست، چه در حالت تحویل به کاربر و چه در حالت drop شدن، آزاد شود.

۱-۳ ساختار گزارش

ساختار این گزارش مطابق با گام‌های تعریف‌شده در صورت پروژه تنظیم شده است. در هر بخش، توضیح پیاده‌سازی همان گام در کد ارائه شده و روند تکمیل پروژه از درایور کارت شبکه تا پشتیبانی UDP در فضای کاربر به‌صورت مرحله‌به‌مرحله بررسی شده است.

فصل ۲

گام اول: تکمیل بخش‌های اولیه درایور E۱۰۰۰

در گام اول، فایل kernel/e1000.c که بخش اصلی درایور کارت شبکه در آن قرار دارد را کامل کردیم تا زیرساخت ارسال و دریافت packet در سطح kernel آماده شود.

۲-۱ مقداردهی اولیه ring RX

در بخش دریافت، ابتدا RX ring را به‌صورت کامل مقداردهی کردیم تا کارت شبکه هنگام دریافت packet فضای معتبر برای نوشتن داده داشته باشد. برای این کار:

- برای هر خانه از RX ring یک buffer با استفاده از ()kalloc ساختیم و آدرس آن را در فیلد addr همان descriptor قرار دادیم تا کارت بتواند از طریق DMA داده را مستقیم داخل حافظه بنویسد.
- آدرس پایه‌ی RX ring و اندازه‌ی آن را در registerهای مربوطه تنظیم کردیم تا کارت شبکه بداند حلقه در چه بخشی از حافظه قرار دارد.
- مقادیر اولیه‌ی RDH و RDT را تنظیم کردیم تا وضعیت ابتدایی حلقه مشخص باشد و دریافت بتواند شروع شود.

۲-۲ مقداردهی اولیه ring TX

در بخش ارسال هم TX ring را آماده کردیم تا بتواند هایpacket خروجی را مدیریت کند. در این قسمت:

- آدرس پایه‌ی TX ring را در registerهای مربوطه قرار دادیم تا کارت شبکه بتواند descriptorهای ارسال را بخواند.

- طول حلقه را تنظیم کردیم و مقادیر اولیه‌ی TDH و TDT را مشخص کردیم.
- وضعیت اولیه‌ی descriptorهای TX را تنظیم کردیم تا هنگام اولین ارسال، هر خانه در وضعیت مشخصی قرار داشته باشد.

۲-۳ تکمیل تابع transmit_1000e

در تابع transmit_1000e بخش‌هایی را اضافه کردیم تا وقتی network stack درخواست ارسال دارد، packet به‌درستی داخل TX ring قرار بگیرد. در این تابع:

- مقدار فعلی TDT را خواندیم تا اندیس descriptor بعدی مشخص شود.
- وضعیت آن descriptor را بررسی کردیم تا مطمئن شویم هنوز در حال استفاده نیست.
- آدرس buffer ارسالی را در فیلد addr قرار دادیم و مقدار length را تنظیم کردیم.
- بیت‌های کنترلی لازم برای ارسال (مانند EOP و RS) را فعال کردیم.
- در پایان، مقدار TDT را جلو بردیم تا کارت شبکه متوجه شود داده‌ی جدید برای ارسال آماده شده است.

۲-۴ تکمیل تابع recv_1000e

تابع recv_1000e را هم کامل کردیم تا packet های دریافتی از روی RX ring خوانده شوند و به لایه‌ی بالاتر تحویل داده شوند. منطق این تابع به این صورت پیاده‌سازی شد:

- حلقه‌ی RX ring را بررسی کردیم و descriptorهایی که وضعیت دریافت آن‌ها فعال شده بود شناسایی کردیم.
- buffer مربوط به هر packet جدید را به rx_net() تحویل دادیم تا پردازش ادامه پیدا کند.
- بعد از تحویل، یک buffer جدید با kalloc() ساختیم و جایگزین قبلی کردیم تا حلقه برای دریافت‌های بعدی آماده بماند.
- در نهایت RDT را به‌روزرسانی کردیم تا کارت شبکه بتواند دوباره از آن خانه استفاده کند.

فصل ۳

گام دوم: پیاده‌سازی مسیر ارسال (TX)

در گام دوم، ارسال بسته‌های شبکه در درایور کارت شبکه را پیاده کردیم. زمانی که پشته‌ی شبکه در فایل net.c نیاز به ارسال یک packet دارد، با فراخوانی تابع transmit_1000e، پوینتر buffer داده به درایور منتقل می‌شود و درایور آن را داخل TX ring قرار داده و کارت شبکه را مطلع می‌کند.

در ابتدای تابع transmit_1000e، مقدار فعلی register مربوط به TDT خوانده می‌شود تا مشخص شود کدام خانه از TX ring برای ارسال بعدی استفاده خواهد شد. این ایندکس تعیین می‌کند داده‌ی جدید در کدام descriptor قرار می‌گیرد و مدیریت صحیح آن از overwrite شدن descriptorهای در حال استفاده جلوگیری می‌کند. سپس وضعیت descriptor بررسی می‌شود و بیت E1000_TXD_STAT_DD چک می‌شود تا مطمئن شویم ارسال قبلی کامل شده و آن خانه آزاد است.

در ادامه، آدرس buffer ارسالی که در net.c با kalloc تخصیص داده شده در فیلد addr قرار می‌گیرد و طول packet در فیلد length تنظیم می‌شود. سپس بیت‌های کنترلی descriptor مقداردهی می‌شوند؛ بیت EOP برای مشخص کردن پایان packet و بیت RS برای درخواست گزارش وضعیت فعال می‌شود. فعال بودن RS باعث می‌شود کارت شبکه پس از اتمام ارسال، بیت DD را تنظیم کند.

پس از تکمیل descriptor، مقدار TDT یک واحد جلو برده شده و در register نوشته می‌شود. با این کار، کارت شبکه متوجه آماده بودن یک descriptor جدید شده و packet را از طریق DMA ارسال می‌کند.

در نهایت، آزادسازی buffer تنها پس از فعال شدن بیت DD انجام می‌شود و با kfree حافظه آزاد می‌گردد. این ترتیب مانع آزاد شدن زودهنگام حافظه و بروز خطا در ارسال می‌شود.

فصل ۴

گام سوم: پیاده‌سازی مسیر دریافت (RX)

در این گام، منطق دریافت بسته‌ها در درایور کارت شبکه را تکمیل کردیم. زمانی که کارت شبکه یک packet را دریافت می‌کند، آن را از طریق DMA در حافظه‌ای می‌نویسد که آدرس آن در فیلد addr یکی از descriptorهای RX ring قرار دارد. پس از نوشتن داده، کارت شبکه با فعال کردن بیت وضعیت descriptor و ایجاد interrupt، kernel را از ورود داده‌ی جدید مطلع می‌کند. پردازش این داده در تابع recv_1000e انجام می‌شود.

در ابتدای تابع recv_1000e، اندیس بررسی از روی مقدار register مربوط به RDT محاسبه می‌شود. سپس descriptorهای بعدی در RX ring بررسی می‌شوند و در هر مرحله بیت وضعیت چک می‌شود. اگر بیت وضعیت فعال نشده باشد، یعنی داده‌ی جدیدی وجود ندارد و پردازش متوقف می‌شود. در غیر این صورت، buffer مربوط به descriptor و طول داده از فیلدهای addr و length خوانده می‌شود.

در این مرحله buffer دریافتی مستقیماً به تابع rx_net() در فایل net.c تحویل داده می‌شود تا پردازش در لایه‌های بالاتر (Ethernet و IP و سپس UDP) ادامه پیدا کند. این کار باعث می‌شود داده بدون کپی اضافی وارد network stack شود.

پس از تحویل packet، همان slot از RX ring باید دوباره برای دریافت آماده شود. برای این کار، یک buffer جدید با kalloc() تخصیص داده شده و آدرس آن جایگزین قبلی می‌شود. سپس بیت وضعیت descriptor پاک می‌شود و مقدار RDT به‌روزرسانی می‌شود تا کارت شبکه بتواند دوباره از آن خانه استفاده کند.

فصل ۵

گام چهارم: کار با register های کارت شبکه

در این گام، ارتباط مستقیم درایور با register های کنترلی کارت شبکه را کامل کردیم. تا این مرحله TX و ring RX در حافظه به‌درستی کار می‌کردند، اما برای اینکه کارت شبکه متوجه تغییرات شود، لازم بود مقدار بعضی از register ها را به‌صورت دقیق بخوانیم و به‌روزرسانی کنیم. دسترسی به این register ها از طریق آرایه‌ی سراسری regs در فایل e1000.c انجام می‌شود.

در مسیر ارسال، بعد از اینکه در تابع transmit_1000e فیلدهای descriptor مقداردهی شدند، مقدار regs[E1000_TDT] جلو برده شد. این همان لحظه‌ای است که کارت شبکه متوجه می‌شود یک descriptor جدید آماده‌ی ارسال است. اگر این مقدار نوشته نشود، حتی با تنظیم کامل descriptor هم packet ارسال نخواهد شد.

در مسیر دریافت نیز داخل تابع recv_1000e بعد از اینکه packet پردازش شد و buffer جدید جایگزین گردید، مقدار regs[E1000_RDT] به‌روزرسانی شد. این کار باعث می‌شود کارت شبکه بداند آن خانه از ring RX دوباره آماده‌ی استفاده است. اگر RDT تغییر نکند، دریافت در همان موقعیت متوقف می‌شود.

در کل،در بخش‌های زیر با register ها کار داریم:

- خواندن regs[E1000_TDT] برای تعیین محل قرار دادن descriptor جدید در ring TX
- نوشتن مقدار جدید در regs[E1000_TDT] بعد از آماده‌سازی descriptor در transmit_1000e
- استفاده از regs[E1000_RDT] برای مشخص کردن موقعیت بررسی در ring RX داخل recv_1000e
- به‌روزرسانی regs[E1000_RDT] بعد از جایگزینی buffer جدید در مسیر دریافت

فصل ۶

گام پنجم: پیاده‌سازی دریافت UDP در net.c

در این گام، دریافت بسته‌های UDP در فایل kernel/net.c را تکمیل کردیم تا فرایندهای کاربر بتوانند داده‌های دریافتی را از طریق syscall دریافت کنند. تا این مرحله، packet ها از کارت شبکه وارد rx_net() می‌شدند، اما مکانیزمی برای نگهداری و تحویل آن‌ها به user space وجود نداشت.

اول ساختاری برای نگهداری وضعیت هر port اضافه شد. برای هر port، یک صف از packet های دریافتی در نظر گرفته شد تا داده‌ها تا زمان فراخوانی recv در kernel باقی بمانند. برای جلوگیری از مصرف بیش از حد حافظه، برای هر port حداکثر ۱۶ packet در صف نگه داشته می‌شود.

در پیاده‌سازی تابع bind، زمانی که یک process یک port مشخص را ثبت می‌کند، وضعیت آن port در ساختار داخلی net.c علامت‌گذاری می‌شود تا packet های با آن مقصد ذخیره شوند. اگر port قبلاً ثبت شده باشد، فراخوانی با خطا مواجه می‌شود. در مقابل، در unbind وضعیت همان port آزاد می‌شود و دیگر packet جدیدی برای آن ذخیره نخواهد شد.

در ادامه، در مسیر دریافت داخل rx_ip()، بعد از اینکه مشخص شد packet از نوع UDP است، شماره‌ی port مقصد از header استخراج می‌شود. سپس بررسی می‌شود که آیا این port قبلاً توسط bind ثبت شده است یا خیر. اگر ثبت نشده باشد، packet آزاد می‌شود. اگر ثبت شده باشد و صف آن port هنوز ظرفیت داشته باشد، packet در صف قرار می‌گیرد؛ در غیر این صورت drop packet می‌شود تا از پر شدن حافظه جلوگیری شود.

در پیاده‌سازی recv، زمانی که process فراخوانی انجام می‌دهد، اگر در صف مربوط به port داده‌ای موجود باشد، اولین packet از صف برداشته شده و محتوای آن به فضای کاربر کپی می‌شود. پس از تحویل داده، buffer مربوطه با kfree آزاد می‌شود. اگر صف خالی باشد، رفتار مطابق طراحی پروژه انجام می‌شود (مانند بازگشت مقدار مناسب یا انتظار).

فصل ۷

گام ششم: تکمیل پردازش UDP در ip_rx

تمام تغییرات این گام در تابع ip_rx در فایل kernel/net.c انجام شد. این تابع برای هر بسته‌ی IP که از net_rx() وارد سیستم می‌شود فراخوانی می‌شود.

در ابتدای ip_rx، ابتدا طول بسته بررسی می‌شود تا مطمئن شود که حداقل شامل header اترنت و IP است. سپس طول واقعی IP header از روی فیلد ip_vhl برداشته می‌شود تا بتوانیم offset دقیق UDP header را محاسبه کنیم. اگر طول بسته کمتر از مقدار لازم باشد، buffer همان‌جا با kfree آزاد می‌شود.

بعد از تأیید طول، نوع پروتکل بررسی می‌شود و تنها در صورتی که ip_p == IPPROTO_UDP باشد، پردازش ادامه پیدا می‌کند. سپس pointer مربوط به UDP header با استفاده از offset محاسبه‌شده ساخته می‌شود. برای خواندن فیلدهای چندبایتی مانند شماره‌ی پورت و طول، از توابع ntohs() و ntohl() استفاده شده است تا تبدیل از ترتیب بایت شبکه به ترتیب بایت پردازنده به‌درستی انجام شود.

در این بخش، شماره‌ی پورت مقصد استخراج شده و اعتبار آن بررسی می‌شود. اگر پورت خارج از محدوده‌ی تعریف‌شده باشد یا قبلاً توسط bind() ثبت نشده باشد، بسته کنار گذاشته می‌شود. همچنین اگر صف مربوط به آن پورت به حداکثر ظرفیت خود رسیده باشد، بسته drop می‌شود تا از مصرف بیش از حد حافظه جلوگیری شود.

سپس کنترل‌های زیر در ip_rx انجام می‌شود:

- بررسی حداقل طول بسته برای دسترسی امن به header های IP و UDP
- بررسی نوع پروتکل و اطمینان از UDP بودن بسته
- تبدیل ترتیب بایت برای dport، sport و آدرس مبدأ
- کنترل معتبر بودن پورت و ظرفیت صف آن

در صورت معتبر بودن شرایط، بسته در صف مربوط به آن پورت ذخیره می‌شود. (ساختار صف در

ابتدای فایل net.c تعریف شده است.) داخل ip_rx پس از گرفتن lock مربوط به آن پورت، محل ذخیره با استفاده از ایندکس w % UDPQSIZE تعیین می‌شود. سپس اطلاعات بسته شامل آدرس مبدأ، پورت مبدأ و طول payload تنظیم می‌شود. طول payload از مقدار udp->ulen استخراج شده و اندازه‌ی header UDP از آن کم می‌شود. داده‌ی payload با memmove داخل آرایه‌ی داخلی صف کپی می‌شود و ایندکس نوشتن زیاد می‌شود. در نهایت با ()wakeup، فرایندهایی که در ()recv منتظر داده بوده‌اند بیدار می‌شوند.

پس از پایان این مراحل، buffer اولیه‌ی دریافت‌شده با kfree آزاد می‌شود تا از نشت حافظه جلوگیری شود.

فصل ۸

گام هفتم: مدیریت آزادسازی های‌buffer شبکه

در گام آخر نیز مدیریت کامل آزادسازی های‌buffer شبکه را بررسی و تکمیل کردیم تا از نشت حافظه و پر شدن تدریجی حافظه‌ی kernel جلوگیری شود.

در مسیر دریافت، buffer اولیه‌ای که از تابع recv_1000e به net_rx() و سپس به ip_rx() منتقل می‌شود، در نهایت داخل ip_rx() آزاد می‌گردد. در صورتی که بسته به هر دلیل معتبر نباشد(مثل ناکافی بودن طول، غیر UDP بودن پروتکل، ثبت‌نشده بودن port مقصد یا پر بودن صف مربوطه) buffer همان‌جا با kfree(buf) آزاد می‌شود و دیگر وارد صف نمی‌شود. این کار باعث می‌شود حتی در حالت drop شدن بسته‌ها نیز حافظه آزاد شود.

در حالتی که بسته معتبر باشد و در صف مربوط به port ذخیره شود، داده‌ی payload داخل ساختار داخلی صف کپی می‌شود و سپس buffer اصلی که از درایور دریافت شده بود آزاد می‌شود. در نتیجه صف داخلی هر port فقط دیتای مورد نیاز را نگه می‌دارد و وابسته به buffer اولیه باقی نمی‌ماندو خب باعث می‌شود طول عمر buffer های تخصیص‌یافته توسط درایور کوتاه باشد و مدیریت حافظه ساده‌تر شود.

در سمت کاربر، هنگام اجرای recv()، پس از آنکه داده از صف مربوط به port برداشته و به فضای کاربر کپی شد، صف به‌روزرسانی می‌شود و آن خانه از صف دوباره قابل استفاده می‌گردد. چون داده قبلاً کپی شده است، نیازی به نگه‌داشتن buffer اضافی وجود ندارد و هیچ pointer ای به حافظه‌ی آزادشده باقی نمی‌ماند.

در مسیر ارسال نیز آزادسازی buffer تنها پس از اطمینان از اتمام ارسال انجام می‌شود. همان‌طور که در تابع transmit_1000e پیاده‌سازی شده، بعد از فعال شدن بیت وضعیت DD، buffer مربوط به آن descriptor با kfree آزاد می‌شود. این ترتیب باعث می‌شود حافظه پیش از اتمام استفاده‌ی سخت‌افزار آزاد نشود و در عین حال پس از پایان ارسال نیز بدون تأخیر آزاد گردد.

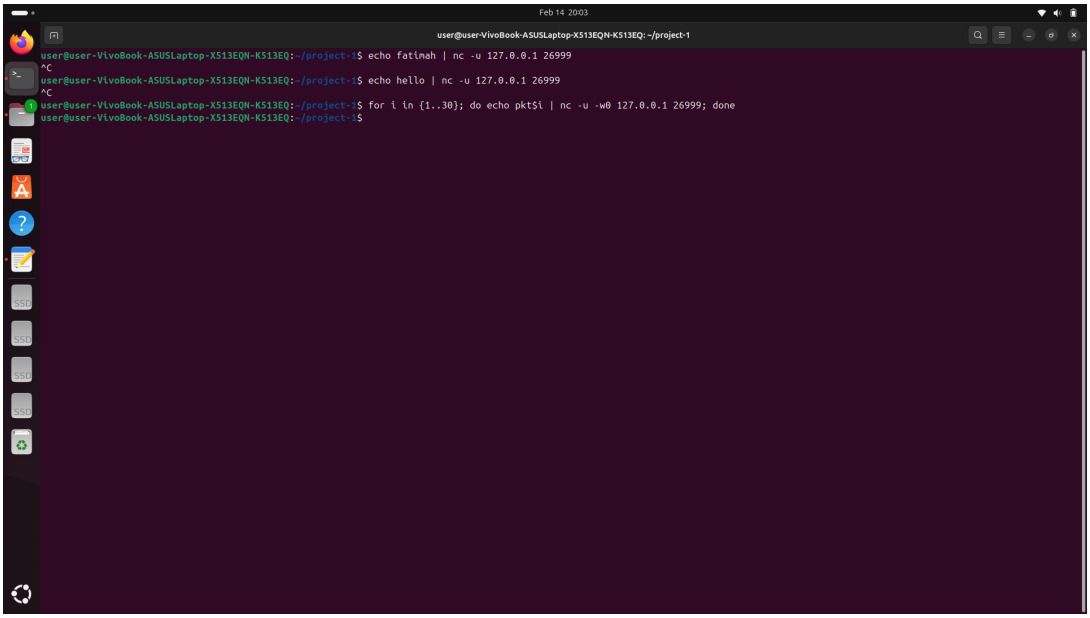
فصل ۹

سناریوی تست شبکه

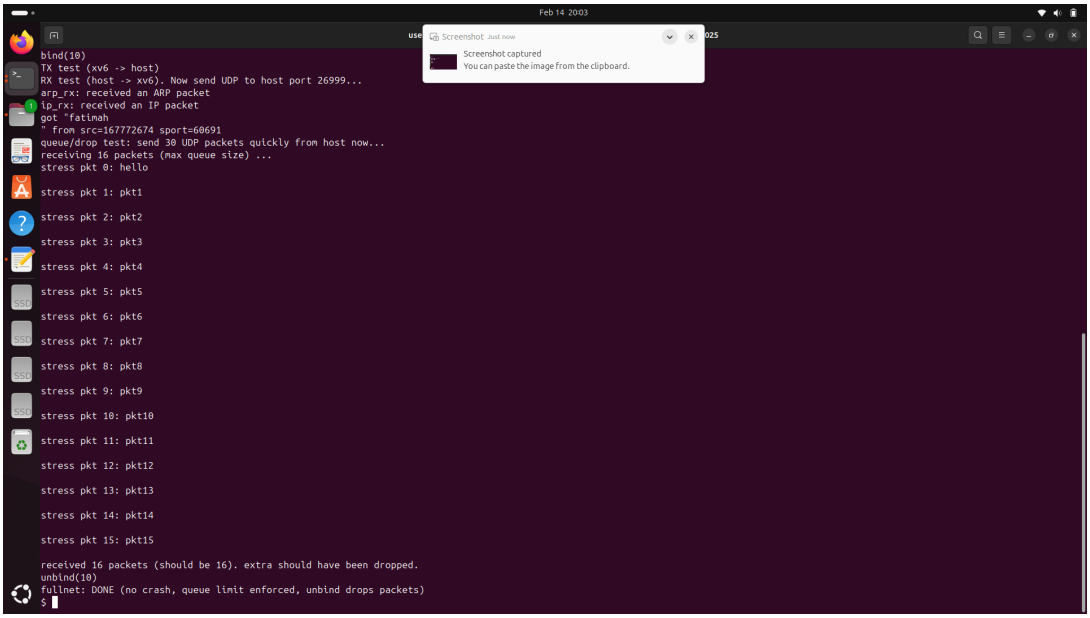
برای تست نهایی مسیر شبکه، یک فایل TestScenario اضافه کردیم تا عملکرد UDP و محدودیت صف را بررسی کنیم. ابتدا در محیط xv6 روی یک port مشخص عملیات bind انجام می‌شود. سپس از سمت host و خارج از فولدر xv6، با استفاده از netcat چند پیام UDP به همان port ارسال می‌کنیم.

در خروجی داخل xv6 مشاهده می‌شود که بسته‌ها توسط ip_rx دریافت شده‌اند و داده‌ها از طریق recv() خوانده می‌شوند. در مرحله‌ی بعد، برای تست محدودیت صف، ۳۰ پیام UDP پشت سر هم ارسال می‌کنیم. طبق پیاده‌سازی، حداکثر ۱۶ بسته در صف هر port نگه داشته می‌شود؛ بنابراین در خروجی دقیقاً ۱۶ پیام دریافت می‌شود و پیام‌های اضافی drop می‌شوند. پیام انتهایی نیز نشان می‌دهد که صف درست مدیریت شده و برنامه بدون crash اجرا را تمام کرده است.

(در ادامه تصاویر ترمینال host و ترمینال داخل xv6 آورده شده‌اند که ارسال بسته‌ها و دریافت ۱۶ پیام و drop شدن بقیه را نشان می‌دهند.)



شکل ۹-۱: ارسال بسته‌های UDP از سمت host برای تست دریافت و محدودیت صف



شکل ۹-۲: خروجی پس از دریافت بسته‌های UDP

فصل ۱۰

بخش امتیازی: پشتیبانی از ICMP و گزارش خطا

در این بخش، برای بهبود رفتار پشته‌ی شبکه در مواجهه با خطاهای ارتباطی، پشتیبانی از پیام‌های ICMP به سیستم اضافه شد. در پیاده‌سازی اولیه، اگر بسته‌ای به مقصد نمی‌رسید، فرایند کاربر از این موضوع مطلع نمی‌شد و ممکن بود در ()recv یا هنگام ارسال، به‌صورت نامحدود منتظر بماند. با اضافه کردن پردازش پیام‌های ICMP، سیستم‌عامل می‌تواند خطاهای مربوط به عدم دسترسی به مقصد را تشخیص داده و به فضای کاربر اعلام کند.

تغییرات اصلی در فایل kernel/net.c و داخل تابع net_rx انجام شد. در این تابع، علاوه بر بررسی بسته‌های ARP و IP/UDP، شرطی برای شناسایی بسته‌هایی با پروتکل IPPROTO_ICMP اضافه شد. در صورت تشخیص ICMP، پردازش به تابع جداگانه‌ای هدایت می‌شود که سرآیند ICMP را تحلیل می‌کند.

در پردازش ICMP، نوع پیام بررسی می‌شود و تمرکز اصلی روی پیام‌های نوع ۳ (Destination Unreachable) قرار دارد. این پیام شامل سرآیند IP و هشت بایت ابتدایی بسته‌ی اصلی است. از این اطلاعات، شماره پورت‌های مبدأ و مقصد UDP استخراج می‌شود تا مشخص شود کدام سوکت مسئول ارسال بسته‌ی اولیه بوده است. با استفاده از این اطلاعات، صف مربوط به همان پورت در ساختار udp_queue شناسایی می‌شود.

برای اعلام خطا به فضای کاربر، ساختار udp_queue با یک فیلد وضعیت خطا تکمیل شد. هنگام دریافت پیام Destination Unreachable، این فیلد برای پورت مربوطه فعال می‌شود. در نتیجه، زمانی که فرایند کاربر تابع ()recv یا ()send را فراخوانی می‌کند، به جای انتظار بی‌پایان یا ارسال ناموفق، کد خطای مناسب دریافت می‌کند و از قطع ارتباط مطلع می‌شود.

با این تغییر، در صورت ارسال داده به یک میزبان غیرقابل‌دسترس، خطا در سطح kernel شناسایی شده و به‌صورت کنترل‌شده به برنامه‌ی کاربر منتقل می‌شود.