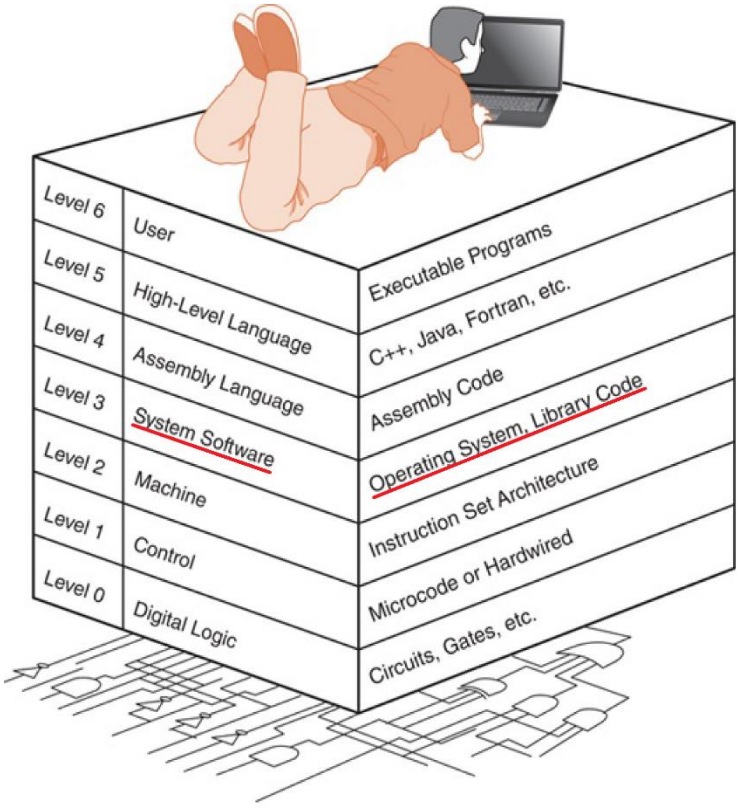
Kolegji UBT

Sistemet Operative
Ligjërata 2

Strukturat e sistemit operativ

MSc. Valdrin Haxhiu



Një sistem operativ ofron ambientin në të cilin ekzekutohen programet. Nga brendia, sistemet operative dallojnë shumë për nga mënyra e ndërtimit, meqë janë të organizuara përgjatë shumë linjave të dizajnit. Dizajnimi i një sistemi operativ është një punë e madhe. Është me rëndësi që qëllimet e sistemit operativ të përcaktohen mirë para se të fillojë dizajnimi. Këto qëllime paraqesin bazën për zgjedhjet në mes algoritmeve dhe strategjive të ndryshme.

Sistemin operativ mund të trajtojmë nga disa vështrime. Njëri vrojtim përqëndrohet në shërbimet që i ofron sistemi operativ; një tjetër, në interfejsin që sistemi operativ ofron për shfrytëzuesit dhe programuesit; një i tretë, në pjesët dhe ndërlidhjet e tyre.

2.1 Shërbimet e sistemit operativ

Një sistem operativ ofron ambientin në të cilin ekzekutohen programet. Ofron disa shërbime për programet dhe shfrytëzuesit e tyre. Shërbimet që ofrohen dallojnë nga njëri sistem operativ në një tjetër, por mund të identifikojmë disa grupe të shërbimeve. Figura 2.1 paraqet një vrojtim për shërbime të ndryshme të sistemit operativ dhe lidhjet në mes tyre.

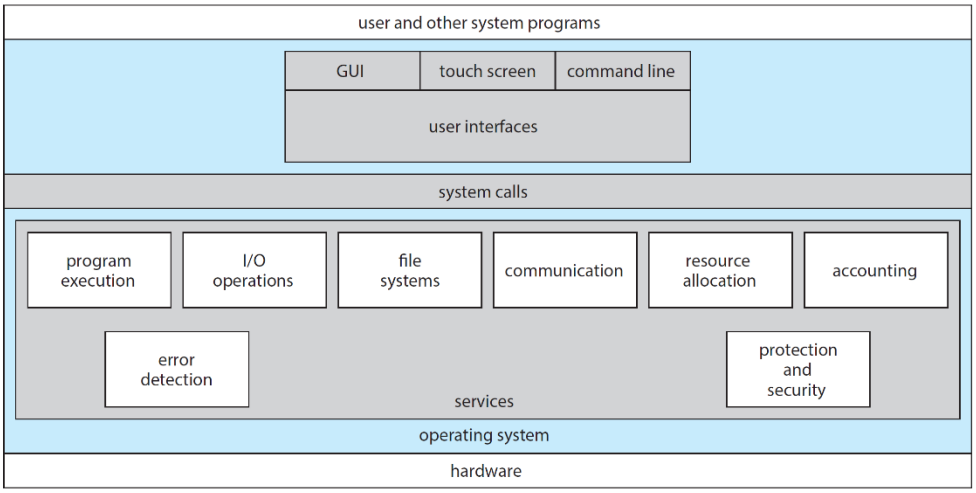


Figura 2.1 Një vrojtim për shërbimet e sistemit operativ

Duhet të kemi parasysh që këto shërbime e bëjnë më të lehtë punën e programuesit.

Një pjesë e shërbimeve të sistemit operativ ofron funksione që ndihmojnë shfrytëzuesin.

- **Interfejsi i shfrytëzuesit**. Pothuajse të gjitha sistemet operative kanë një **interfejs të shfrytëzuesit** (UI). Interfejsi mund të ketë forma të ndryshme. Ky interfejs është një sistem i dritareve me një maus që përdoret si pajisje për të përdorë pajisjet tjera H/D, për të përzgjedhur opcionet nga menytë, për selektime dhe një tastierë për të futur tekst. Sistemet mobile si telefonët dhe tabletët ofrojnë një **interfejs me ekran me prekje**, duke mundësuar që shfrytëzuesit të përdorin gishtërinjtë për të përdorë sistemin dhe shtypur butonat në ekran për përzgjedhjen e opcioneve. Një mundësi tjetër është një **interfejs përmes komandave (CLI)**, që përdorë komanda tekstuale dhe një metodë për futjen e tyre (një tastierë për

shkruarjen e komandave në një format të caktuar me opcione specifike). Disa sisteme ofrojnë dy ose të tre nga këto interfejse.

- **Ekzekutimi i programit**. Sistemi duhet patjetër të jetë në gjendje të vendosë një program në DRAM dhe të ekzekutojë atë. Programi duhet patjetër të ketë mundësinë që të përfundojë ekzekutimin, në mënyrë të rregullt ose në mënyrë jo të rregullt (duke treguar gabimin).
- **Operacionet e hyrje/daljes**. Një progam që është duke u ekzekutuar mund të ketë nevojë për H/D, që mund të përfshijë një fajll ose një pajisje për H/D. Për pajisje të caktuara, mund të jenë të dëshirueshme disa funksione speciale (si për shembull leximi në një kartelë të rrjetës ose shkruarja në një sistem të fajllave). Për efiçiencë dhe mbrojtje, zakonisht shfrytëzuesit nuk mund të kontrollojnë direkt pajisjet H/D. Kështu, sistemi operativ duhet të ofrojë mundësi për të bërë hyrje/dalje.
- **Puna me sistemin e fajllave**. Sistemi i fajllave është i një rëndësie të veçantë. Është e qartë se programet kanë nevojë të lexojnë dhe të shkruajnë fajlla dhe direktoriume. Programet kanë poashtu nevojë të krijojnë dhe fshijnë fajlla përmes emrit të tyre, të kërkojnë për ndonjë fajll të caktuar dhe të listojnë informatat e fajllit. Në fund, disa sisteme operative përfshijnë menaxhimin e të drejtave për të lejuar ose jo qasjen në fajlla dhe direktoriume bazuar në të drejtat mbi fajllin. Shumë sisteme operative ofrojnë disa sisteme të fajllave, në disa për të mundësuar zgjedhjen sipas dëshirës dhe ndonjë herë për të ofruar karakteristika specifike ose karakteristika të performancës.
- **Komunikimet**. Ka shumë rrethana kur një proces ka nevojë të shkëmbejë informacion me ndonjë proces tjetër. Një komunikim i tillë mund të ndodhë në mes proceseve që janë duke u ekzekutuar në të njëjtin kompjuter ose në mes të proceseve që janë duke u ekzekutuar në sisteme të ndryshme kompjuterike të lidhura me një rrjetë. Komunikimet mund të bëhen përmes **memories së përbashkët**, ku dy ose më shumë procese lexojnë dhe shkruajnë në një pjesë të përbashkët të memories, ose përmes **bartjes së mesazheve**, ku pakotë e informacionit me një format të paracaktuar barten në mes të proceseve nga sistemi operativ.
- **Gjetja e gabimeve**. Sistemi operativ ka nevojë që vazhdimisht të gjejë dhe të përmirësojë gabimet. Gabimet mund të paraqiten në CPU dhe në harduer të memories (ndonjë gabim në memorie ose ndërprerja e rrymës elektrike), në pajisje H/D (gabim pariteti në një disk, një dështim i lidhjes së rrjetës ose mungesa e letrës në printer) dhe në program të shfrytëzuesit (tejmbushje aritmetike ose një tentim për t'u qasur në një vend jo të lejuar të memories). Për secilin lloj të gabimit, sistemi operativ duhet të ndërmarrë veprimin e duhur për të siguruar përdorim të rregullt dhe konsistent të kompjuterit. Ndonjë herë, nuk ka zgjidhje tjetër veç të ndalojë sistemin. Në disa raste tjera, mund të ndalojë një proces që ka shkaktuar gabim ose të kthejë një kod (numër) gabimi te procesi ashtu që procesi të gjejë atë dhe të përmirësojë.

Një pjesë tjetër e funksioneve të sistemit operativ ekzistojnë jo për të ndihmuar shfrytëzuesin, por për të siguruar punë efikase të vet sistemit. Sistemet me shumë procese mund të arrijnë efikasitetin duke ndarë resurset në mes proceseve të ndryshme.

- **Ndarja e resurseve**. Në kohën kur disa procese janë duke u ekzekutuar, secilit duhet t'i ndahen resurset. Sistemi operativ menaxhon resurse të ndryshme. Disa (si ciklet e CPU-së, memoria DRAM dhe memoriet e fajllave) mund të kenë një kod special të ndarjes, ndërkohë që disa tjera (si pajisjet për H/D) mund të kenë kode më të përgjithshme të marrjes dhe lirimit. Për shembull, për përdorim sa më të mirë të CPU-së, sistemet operative kanë funksione (metoda) për planifikim të CPU-së që marrin parasysh shpejtësinë e CPU-së, procesin që duhet patjetër të ekzekutohet, numrin e bërthamave në CPU dhe faktorë tjerë. Mund të jenë edhe funksione për të përdorë bashkë printerët, pajisjet USB për ruajtje dhe pajisje tjera periferike.
- **Logimi**. Duhet të mbahen shënime se cilat resurse të kompjuterit përdorin cilat programe dhe për sa kohë. Kjo mbajtje e shënimeve mund të përdoret për pagesë (ashtu që shfrytëzuesit mund të detyrohen të paguajnë) ose thjeshtë për të mbledhur statistikat e përdorimit. Statistikat e përdorimit mund të jenë një vegël me vlerë për administruesit e sistemit të cilët duan të rikonfigurojnë sistemin për të përmirësuar shërbimet në kompjuter.
- **Mbrojtja dhe siguria**. Krijuesit (pronarët) e informacionit të ruajtur në një sistem me shumë shfrytëzues ose kompjuter të lidhur në rrjetë mund të kenë nevojë që të kontrollojnë përdorimin e atij informacioni. Kur disa procese të ndara ekzekutohen në mënyrë konkurente, nuk duhet të jetë e mundur që një proces të përzihet në punët e proceseve tjera apo edhe në punët e vet sistemit operativ. Mbrojtja siguron që të gjitha qasjet në resurset e kompjuterit bëhen në mënyrë të kontrolluar. Siguria e sistemit nga palët jashtë kompjuterit është poashtu me rëndësi. Një siguri e tillë fillon duke kërkuar që çdo shfrytëzues të autentikojë veten e tij ose saj në sistem, zakonisht përmes një fjalëkalimi (password) për t'u qasur resurseve të sistemit. Kjo zgjerohet edhe në nevojën për mbrojtjen e pajisjeve H/D, përfshirë kartelat e rrjetës, nga tentimet jo të lejuara për qasje dhe ruajtja e të gjitha atyre lidhjeve për të gjetur ndonjë thyerje të sigurisë. Për të qenë një sistem i mbrojtur dhe i sigurtë, masat parandaluese duhet të jenë të vendosura brenda tij. Një zinxhirë është i fortë aq sa është e fortë pjesa më e dobët e tij.

2.2 Interfejsi i shfrytëzuesit dhe sistemit operativ

Ekzistojnë disa mënyra për shfrytëzuesit që të përdorin sistemin operativ. Janë tre qasje themelore. Njëra ofron një interfejs përmes komandave, ose **interpretues të komandave**, që lejon shfrytëzuesit që të fusin direkt komanda për t'u ekzekutuar nga sistemi operativ. Dy qasjet tjera ia mundësojnë shfrytëzuesit të përdorë sistemin operativ përmes interfejsit grafik të shfrytëzuesit ose GUI.

2.2.1 Interpretuesit e komandave

Shumica e sistemeve operative, përfshirë Linux, UNIX dhe Windows, interpretuesin e komandave e trajtojnë si një program të veçantë që ekzekutohet kur të fillojë një proces ose kur një shfrytëzues logohet për herë të parë në sistem (në sistemet interaktive). Në sistemet me shumë interpretues të komandave, interpretuesit njihen si **shell**. Për shembull, në sistemet UNIX dhe Linux, një shfrytëzues mund të zgjedhë në mes disa shell-eve, përfshirë shell-in C, Bourne-Again shell, Korn shell dhe të tjerë. Shumica e shell-eve ofrojnë funksionet e njëjta, shfrytëzuesit

përzgjedhin njërin shell për përdorim varësisht prej preferencës së tyre. Figura 2.2 paraqet shell-in Bourne-Again (ose bash) interpretues të komandave në sistemin operativ macOS.

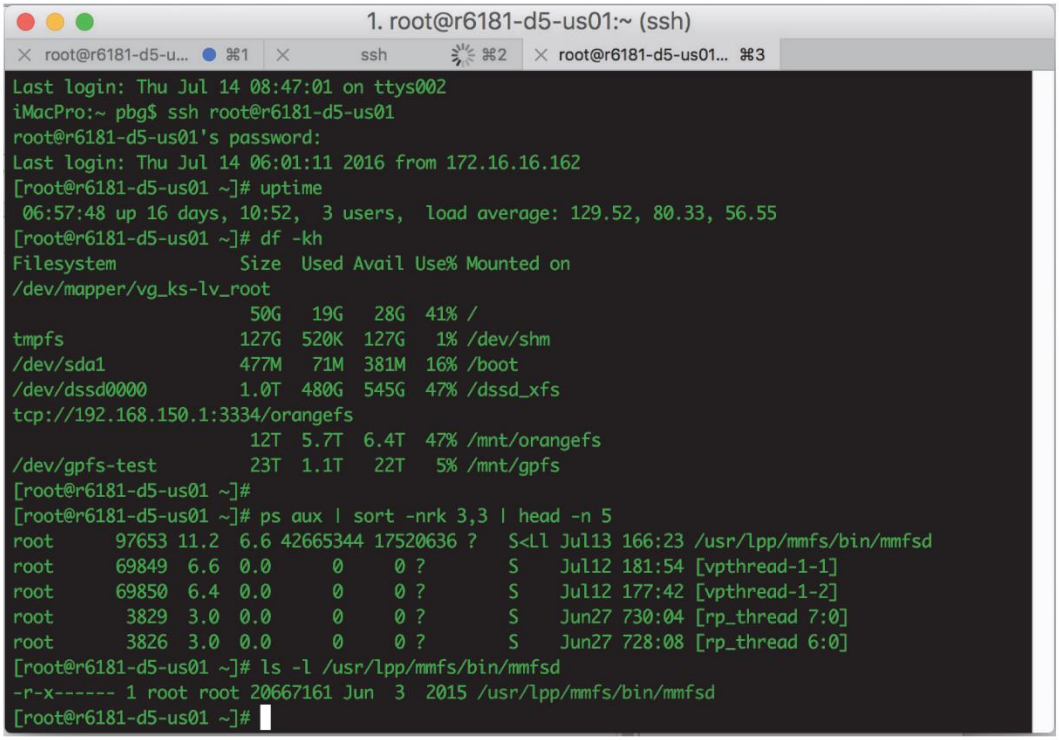


Figura 2.2 Interpretuesi i komandave bash në macOS

Funksioni kryesor i një interpretuesi të komandave është të lexojë dhe ekzekutojë komandën e radhës të dhënë nga shfrytëzuesi. Shumë nga komandat të dhëna në këtë nivel punojnë me fajlla: krijo (create), fshij (delete), listo (list), printo (print), kopjo (copy), ekzekuto (execute) e tjera. Shumë shell-e në sistemet UNIX punojnë në këtë mënyrë. Këto komanda mund të realizohen në dy mënyra të përgjithshme.

Në njërën prej qasjeve, vet interpretuesi i komandave përmban kodin për ekzekutimin e komandës. Për shembull, një komandë për fshirjen e një fajlli mund të bëjë që interpretuesi i komandave të kërcejë te një pjesë e kodit të tij që bën gati parametrat dhe thirrë thirrjen e duhur sistemore. Në këtë rast, numri i komandave që mund t'i jepen përcakton madhësinë e interpretuesit të komandave, meqë secila komandë kërkon kodin e vet për implementim.

Një qasje tjetër – që përdoret nga UNIX, në mesin e sistemeve operative tjera – implementon shumicën e komandave përmes programeve të sistemit. Në këto raste, interpretuesi i komandave nuk i kupton komandat; thjeshtë e përdorë komandën për të identifikuar një fajll për të vendosur në memorie dhe për të ekzekutuar. Kështu, komanda në UNIX për fshirjen e një fajlli

```
rm file.txt
```

do të kërkonte programin (fajllin) me emrin `rm`, vendosë fajllin në memorie dhe ekzekutojë atë me parametrin `file.txt`. Logjika e programit të komandës `rm` do të mund të përcaktohej plotësisht me kod në fajllin `rm`. Në këtë mënyrë, programuesit mund të shtojnë lehtë komanda të reja në sistem duke krijuar fajlla të rinjë duke vendosur brenda tyre logjikën e duhur të programit. Programi i interpretuesit të komandave, që mund të jetë i vogël, nuk është e nevojshme të ndryshohet kur të shtohen komanda të reja.

2.2.2 Interfejsi grafik i shfrytëzuesit

Një mënyrë tjetër e përdorimit të sistemit operativ është përmes interfejsit grafik të shfrytëzuesit. Përmes kësaj mënyreje, në vend të futjes direkt të komandave në një interpretues të komandave, shfrytëzuesi përdorë një sistem që ka dritare dhe meny përmes një mausi dhe një **desktop**. Shfrytëzuesi lëvizë mausin për të vendosur kursorin mbi foto, ose **ikona** në ekran (desktop) që paraqesin programe, fajlla, direktoriume dhe funksione të sistemit. Varësisht prej pozitës së kursorit të mausit, shtypja (klikimi) e një butoni në maus mund të startojë një program, përzgjedhë (selektojë) një fajll ose direktorium – i njohur si follder (folder) – ose të hapë një meny që përmban komanda.

Interfejsët grafik të shfrytëzuesit u paraqitën për herë të parë gjatë kërkimeve që u bënë në qendrat për kërkim të Xerox PARC në fillim të viteve të 1970-ta. GUI i parë u paraqit në kompjuterin Xerox Alto në vitin 1973. Megjithatë, interfejsët grafik u përhapën më shumë me ndërtimin kompjuterëve Apple Macintosh në vitet e 1980-ta. Interfejsi i shfrytëzuesit të sistemit operativ Macintosh ka pësuar shumë ndryshime gjatë viteve, ndryshimi më i madh ishte interfejsi ***Aqua*** në sistemin macOS. Versioni i parë i Windows-it të Microsoft-it – versioni 1.0 – ishte i bazuar në sistemin operativ MS-DOS duke ia shtuar një interfejs GUI. Versionet e mëvonshme të Windows-it bënë ndryshime të mëdha në GUI bashkë me disa përmirësime në funksionalitet.

Tradicionalisht, sistemet UNIX kanë qenë të dominuara nga interfejsi me komanda. Ekzistojnë disa interfejse GUI, megjithatë, me zhvillimet e mëdha në dizajnët e GUI nga disa projekte open-source (me kod të hapur), si ***K Desktop Environment*** (ose *KDE)* dhe desktop GNOME nga projekti GNU. Desktopët KNE dhe GNOME ekzekutohen në Linux dhe disa sisteme UNIX dhe janë në dispozicion me licenca open-source, që nënkupton që kodi i tyre burimor është në dispozicion për lexim dhe për ndryshim nën kushte specifike të licencës.

2.2.3 Interfejsi me ekran me prekje

Shumica e sistemeve mobile, telefonët e mençur dhe tabletët kanë një interfejs me ekran me prekje sepse një interfejs me komanda ose interfejs maus dhe tastierë nuk është i përshtatshëm. Në këto sisteme, shfrytëzuesit përdorin sistemin me anë të veprimeve me gishtërinjë në ekranin me prekje – për shembull, shtypja dhe rrëshqitja e gishtërinjëve në ekran. Edhepse gjeneratat e mëhershme të telefonëve të mençur kishin një tastierë, tash shumica e telefonëve dhe tabletëve e simulojnë (përmes softuerit) tastierën në ekran. Figura 2.3 paraqet pamjen e ekranit me prekje në në Apple iPhone. iPad dhe iPhone përdorin interfejsin me prekje **Springboard**.

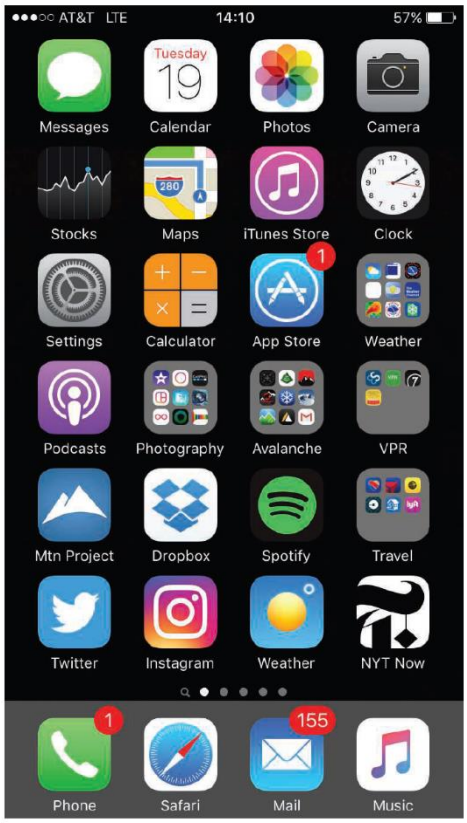


Figura 2.3 Ekrani me prekje në iPhone

2.2.4 Zgjedhja e interfejsit

Zgjedhja për të përdorë një interfejs me komanda ose një interfejs GUI është kryesisht çështje e preferencës personale. **Administruesit e sistemit** që menaxhojnë sistemet dhe shfrytëzuesit me shumë njohuri (**power users**) për sistemin shpesh e shfrytëzojnë interfejsin me komanda. Për ta, është më efikase, duke ua mundësuar atyre një qasje më të shpejtë në veprimet që ata duan të kryejnë. Në të vërtetë, në disa sisteme, vetëm një pjesë e funksioneve të sistemit ofrohet përmes GUI, duke lënë për shfrytëzuesit që kanë më shumë njohuri për sistemin veprimet që nuk janë të zakonshme. Për më tepër, interfejset përmes komandave zakonisht i bëjnë punët që janë të përsëritshme më të lehta, pjesërisht për arsye të programueshmërisë që kanë. Për shembull, nëse një punë kërkon një bashkësi të hapave përmes komandave, ata hapa mund vendosen në një fajll dhe ai fajll mund të ekzekutohet si një program. Programi nuk kompajlohet në një kod të ekzekutueshëm, por interpretohet nga interfejsi i komandave. Këto skripta shell janë shumë të zakonshme në sistemet që janë të orientuara në komanda, si UNIX dhe Linux.

Për dallim, shumica e shfrytëzuesve të Windows-it janë të kënaqur të përdorin ambientin GUI në Windows dhe pothuajse nuk përdorin kurrë interfejsin përmes shell-it. Versionet e fundit të sistemit operativ Windows ofrojnë një interfejs GUI për sistemet desktop dhe laptop dhe një interfejs përmes ekranit me prekje në tabletë. Shumë ndryshime që janë bërë në sistemet operative Macintosh ofrojnë poashtu një rast studimi si dallim. Historikisht, macOS nuk ka ofruar një interfejs përmes komandave, duke kërkuar nga shfrytëzuesit të përdorin sistemin

operativ vetëm përmes GUI. Megjithatë, përmes macOS (që pjesërisht është implementuar duke përdorë një kernel UNIX), sistemi operativ ofron një interfejs Aqua GUI dhe një interfejs me komanda. Figura 2.4 paraqet një pamje të GUI në macOS.

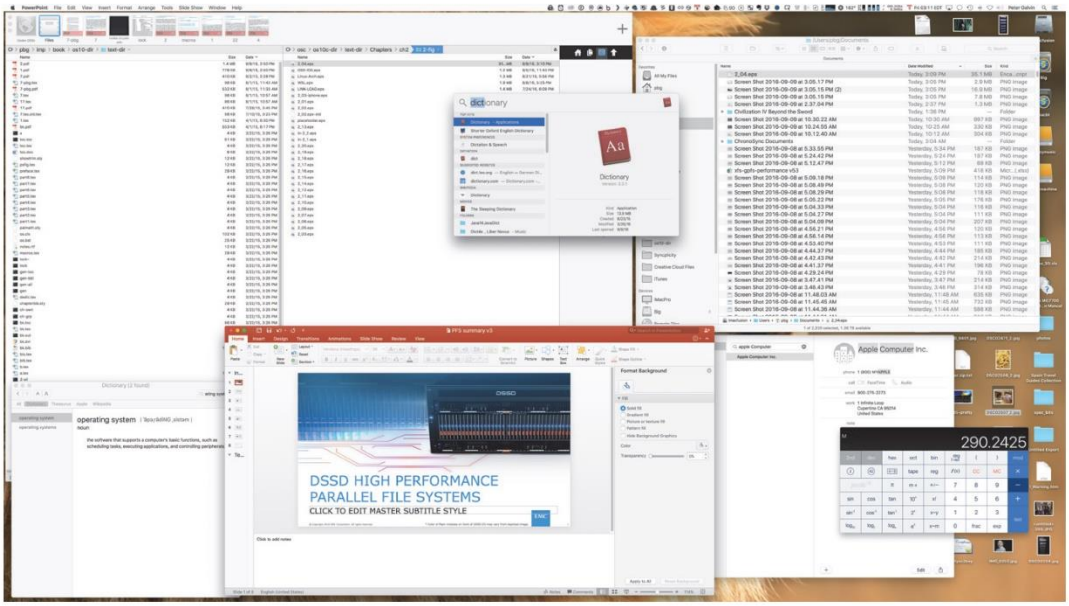


Figura 2.4 GUI në macOS

Edhepse ka programe që ofrojnë interfejs me komanda për iOS dhe sistemet mobile Android, ato përdoren rrallë. Shumica e shfrytëzuesve të sistemeve mobile përdorin pajisjet mobile përmes interfejsit ekran me prekje.

Interfejsi i shfrytëzuesit mund të ndryshojë nga një sistem në një tjetër madje edhe nga shfrytëzuesi në shfrytëzues në një sistem; megjithatë, zakonisht largohet nga struktura e sistemit. Dizajni i një interfejsi lehtë të përdorshëm për shfrytëzuesin nuk është një funksion direkt i sistemit operativ.

2.3 Thirrjet sistemore

Thirrjet sistemore ofrojnë një interfejs (mundësi) për përdorimin e shërbimeve që ofrohen nga një sistem operativ. Këto thirrje zakonisht janë në formë të funksioneve të shkruara në gjuhët C dhe C++, edhepse disa punë të nivelit më të ulët (për shembull, punët ku hardueri duhet të qaset direkt) mund të jetë e nevojshme të shkruhen përmes instruksioneve në gjuhën asembler.

2.3.1 Shembull

Para se të flasim për atë se si i ofron thirrjet sistemore një sistem operativ, të shohim së pari një shembull për të treguar si përdoren thirrjet sistemore: shkruarja e një programi për leximin e të dhënave nga një fajll dhe kopjimin e tyre në një fajll tjetër. Hyrja (inputi) e parë që i duhet programit janë emrat e të dy fajllave: fajlli hyrje dhe fajlli dalje. Këta emra mund të jepen në

shumë mënyra, varësisht prej dizajnit të sistemit operativ. Një qasje është që emrat e të dy fajllave të barten si pjesë të komandës – për shembull, komanda `cp` në UNIX:

```
cp in.txt out.txt
```

Kjo komandë e kopjon përmbajtjen e fajllit `in.txt` në fajllin dalës `out.txt`. Një qasje tjetër është që programi të kërkojë emrat e fajllave prej shfrytëzuesit. Në një sistem interaktiv, kjo qasje do të kërkojë një radhë të thirrjeve sistemore, së pari për paraqitjen e një mesazhi kërkues për shfrytëzuesin (që shfrytëzuesi të japë emrat e fajllave) në ekran dhe pastaj leximin përmes tastierës të karaktereve që definojnë të dy fajllat. Në sistemet që përdorin maus dhe ikona, zakonisht paraqitet një meny me emra të fajllave në një dritare. Shfrytëzuesi mund të përdorë mausin për zgjedhjen e fajllit të parë (burim) dhe një dritare mund të hapet për zgjedhjen e emrit të fajllit destinacion. Kjo radhë kërkon shumë thirrje sistemore për H/D.

Kur të jenë marrë emrat e fajllave, programi duhet patjetër të hapë fajllin hyrës dhe të krijojë e të hapë fajllin dalës. Secili nga këto operacione kërkon një thirrje sistemore tjetër. Ndonjë gabim eventual që lidhet me thirrjet sistemore duhet të trajtohet patjetër. Për shembull, kur programi të tentojë të hapë fajllin hyrës, mund të kuptojë që nuk ka një fajll me atë emër ose fajlli mund të jetë i mbrojtur sa i përket qasjes së tij. Në këto raste, programi duhet të paraqesë një mezash gabimi (një radhë tjetër e thirrjeve sistemore) dhe të përfundojë ekzekutimin në mënyrë jo të rregullt (një thirrje tjetër sistemore). Nëse fajlli hyrës ekziston, duhet patjetër të krijohet një fajll i ri dalës. Mund të ekzistojë një tjetër fajll dalës me emrin e njëjtë. Kjo situatë mund të bëjë që programi të ndalë ekzekutimin (një thirrje sistemore) ose mund të fshijmë fajllin dalës ekzistues (një thirrje sistemore tjetër) dhe të krijojë një fajll të ri (një thirrje sistemore tjetër). Një mundësi tjetër, në një sistem interaktiv, është që të kërkohet prej shfrytëzuesit (me një radhë të thirrjeve sistemore për të paraqitur një mesazh kërkues dhe të lexohet përgjigjja prej terminalit) për të zëvendësuar fajllin ekzistues ose për të ndalur programin.

Kur të jenë rregulluar të dy fajllat, programi futet në një loop (unazë që ekzekuton me përsëritje disa herë disa instruksione) që lexon prej fajllit hyrës (një thirrje sistemore) dhe shkruan në fajllin dalës (një thirrje sistemore tjetër). Secili shkrim dhe lexim duhet patjetër të kthejë një informacion të statusit në lidhje me gabime të mundshme. Gjatë leximit në fajllin hyrës, programi mund të kuptojë se ka arritur në fund të fajllit ose është paraqitur ndonjë dështim në harduer (ndonjë gabim pariteti). Operacioni i shkrimit mund të hasë në gabime të ndryshme, varësisht prej pajisjes dalëse (për shembull, nuk ka hapësirë të lirë në disk).

Në fund, kur i tërë fajlli të jetë kopjuar, programi mund të mbyllë të dy fajllat (dy thirrje sistemore), shkruaj një mesazh në ekran (konzolë) ose dritare (më shumë thirrje sistemore) dhe në fund të ndalojë programin në mënyrë të rregullt (thirrja sistemore e fundit). Kjo radhë e thirrjeve sistemore është paraqitur në Figura 2.5.

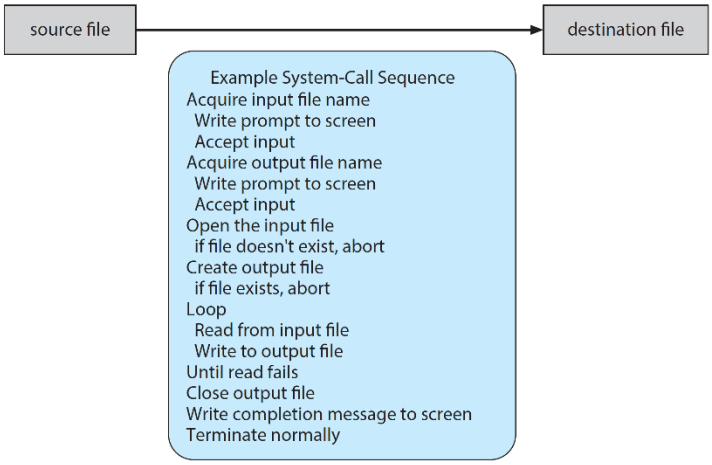


Figura 2.5 Shembull se si përdoren thirrjet sistemore

2.3.2 Interfejsi i programimit (application programming interface)

Edhe programet më të thjeshtë mund të përdorin shumë sistemin operativ. Shpesh, sistemet ekzekutojnë mijëra thirrje sistemore brenda një sekonde. Shumica e programuesve nuk e shohin këtë nivel të detajeve. Zakonisht, zhvilluesit e aplikacioneve dizajnojnë programet sipas një **interfejsi të programimit** (**API**). API specifikon një bashkësi të funksioneve që janë në dispozicion për një programues të aplikacioneve, përfshirë parametrat që i barten secilit funksion dhe vlerat e kthimit që mund të presë programuesi. Tre prej API-ve më të zakonshëm që janë në dispozicion të programuesve të aplikacioneve janë Windows API për sistemet Windows, POSIX API për sistemet Posix (që përfshijnë pothuajse të gjitha versionet e UNIX, Linux dhe macOS) dhe Java API për programet që ekzekutohen në makinën virtuale të Java. Një programues i qaset një API përmes një librarie me kod që ofrohet nga sistemi operativ. Në rastin e UNIX dhe Linux për programet e shkruara në C, libraria quhet **libc**. Secili sistem operativ ka emrin e tij për secilën thirrje sistemore.

Funksionet brenda një API zakonisht i thërrasin thirrjet sistemore për programuesin e aplikacioneve. Për shembull, funksioni CreateProcess ( ) në Windows (që përdoret për të krijuar një proces të ri) e thërret thirrjen sistemore NTCreateProcess ( ) në kernelin e Windows-it.

Pse një programues i aplikacioneve do të preferonte programimin përmes API se sa përmes thirrjes direkte të thirrjeve sistemore? Janë disa arsye. Njëri përfitim ka të bëjë me mundësinë e bartjes së kodit të programit. Një programuese e aplikacioneve që dizajnon një program duke përdorë një API mund të presë që programi i saj të kompajlohet dhe të ekzekutohet në cilindo sistem që përkrahë të njëjtin API (megjithatë, në realitet, dallimet në arkitekturë shpesh e bëjnë këtë më të vështirë se sa mund të duket). Për më tepër, thirrjet sistemore mund të jenë më të detajuara dhe më të vështira për të punuar me to se sa API që është në dispozicion të programuesit të aplikacionit. Megjithatë, zakonisht ekziston një lidhje e fortë në mes të një funksioni në API dhe thirrjes së vet sistemore në kernel. Në fakt, shumë nga API-të në POSIX dhe Windows janë të ngjashme me thirrjet sistemore të ofruara nga sistemet operative UNIX, Linux dhe Windows.

Një faktor tjetër me rëndësi në trajtimin e thirrjeve sistemore është rrethina në **kohën e ekzekutimit (run-time environment – RTE)** – e tërë bashkësia e softuerëve që nevojitet për të ekzekutuar programet e shkruara në një gjuhë programuese, përfshirë kompajlerët ose interpretuesit si dhe softuerë tjerë, si libraritë dhe loaders. RTE ofron një **system-call interface** (**interfejs për thirrjet sistemore**) që shërben si lidhje për te thirrjet sistemore që ofrohen nga sistemi operativ. Interfejsi për thirrjet sistemore pranon thirrjet e funksioneve në API dhe thërret thirrjet sistemore të nevojshme brenda në sistem operativ. Zakonisht, secila thirrje sistemore e ka numrin e saj dhe interfejsi për thirrjet sistemore e mirëmban një tabelë të indeksuar me numra të thirrjeve sistemore. Interfejsi për thirrjet sistemore thërret thirrjen sistemore të nevojitur në kernelin e sistemit operativ dhe kthen statusin e thirrjes sistemore.

Thirrësi i thirrjes sistemore nuk ka nevojë të di asgjë sa i përket implementimit (kodit) të thirrjes sistemore ose çfarë bën ajo gjatë ekzekutimit. Thirrësi ka nevojë të di si të përdorë API dhe të kuptojë se çfarë rezultati do të arrihet kur të ekzekutohet ajo thirrje sistemore. Kështu, shumica e detajeve të interfejsit të sistemit operativ fshehen nga programuesi përmes API dhe menaxhohen nga RTE. Lidhja në mes një API, interfejsit të thirrjes sistemore dhe sistemit operativ është paraqitur në Figura 2.6, që tregon se si sistemi operativ e trajton një program të shfrytëzuesit që thërret thirrjen sistemore open ( ).

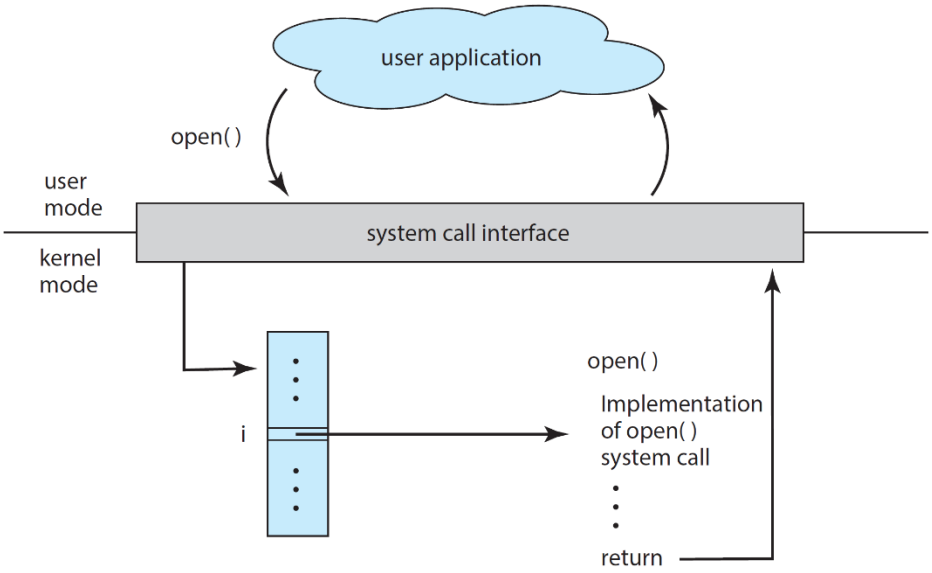


Figura 2.6 Trajtimi i thirrjes së thirrjes sistemore open ( ) nga një program i shfrytëzuesit

Thirrjet sistemore ndodhin në mënyra të ndryshme varësisht prej kompjuterit në përdorim. Shpesh, më shumë informacion nevojitet se sa vetëm identiteti i thirrjes sistemore të dëshiruar. Lloji i saktë dhe sasia e informacionit dallojnë sipas sistemit operativ të caktuar dhe thirrjes. Për shembull, për leximin e hyrjes, mund të kemi nevojë të specifikojmë fajllin ose pajisjen për përdorim si burim, si dhe adresën dhe gjatësinë e baferit të memories në të cilin lexohet hyrja. Sigurisht, pajisja ose fajlli dhe gjatësia mund të jenë të nënkuptueshme në thirrje.

Mund të përdoren tre forma të ndryshme të bartjes së parametrave te sistemi operativ. Qasja më e thjeshtë është që parametrat të vendosen në regjistra. Në disa raste, megjithatë, mund

të ketë më shumë parametra se sa regjistra. Në këto raste, parametrat zakonisht ruhen në një bllok ose tabelë në memorie dhe adresa e bllokut (ose tabelës) vendoset si parametër në një regjistër (Figura 2.7).

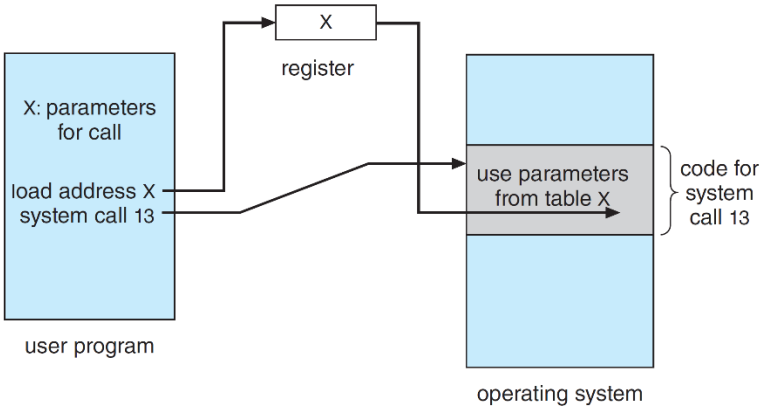


Figura 2.7 Bartja e parametrave përmes një tabele

Linux përdorë një kombinim të këtyre qasjeve. Nëse janë pesë ose më pak parametra, përdoren regjistrat. Nëse janë më shumë se pesë parametra, përdoret metoda përmes bllokut. Parametrat mund të vendosen edhe në **stack (stek)** nga programi dhe të lexohen prej stekut nga sistemi operativ. Disa sisteme operative përdorin metodën përmes bllokut ose stekut sepse këto qasje nuk e kufizojnë numrin ose gjatësinë e parametrave që barten.

2.3.3 Llojet e thirrjeve sistemore

Thirrjet sistemore mund të ndahen në gjashtë kategori kryesore: **kontroll i procesit, menaxhim i fajllave, menaxhim i pajisjeve, mirëmbajtje e informacionit, komunikime** dhe **mbrojtje**. Figura 2.8 përmbledhë llojet e thirrjeve sistemore që mund të ofrohen nga një sistem operativ.

- Kontroll procesi
  - krijo procesin, përfundo procesin
  - vendose në memorie, ekzekuto
  - lexo atributet e procesit, vendos atributet e procesit
  - ngjarje e pritjes, ngjarje e sinjalit
  - dhënia dhe lirimi i memories
- Menaxhimi i fajllave
  - krijo fajllin, fshij fajllin
  - hape, mbylle
  - lexo, shkruaj, rivendose
  - lexo atributet e fajllit, vendos atributet e fajllit
- Menaxhimi i pajisjeve
  - merr në shfrytëzim pajisjen, liro pajisjen
  - lexo, shkruaj, rivendose
  - lexo atributet e pajisjes, vendos atributet e pajisjes
  - vendos ose largo pajisjet prej sistemit
- Mirëmbajtja e informacionit

    - lexo kohën ose datën, vendos kohën ose datën
    - lexo të dhënat e sistemit, vendos të dhënat e sistemit
    - lexo atributet e procesit, fajllit ose pajisjes
    - vendos atributet e procesit, fajllit ose pajisjes
- Komunikimet
    - krijo, fshij lidhjen e komunikimit
    - dërgo, prano mesazhet
    - bart informacionin e statusit
    - shto ose largo pajisjet në distancë
- Mbrojtja
    - lexo të drejtat mbi fajllin
    - vendos të drejtat mbi fajllin

Figura 2.8 Llojet e thirrjeve sistemore

2.3.3.1 Kontroll procesi

Një program që është duke u ekzekutuar ka nevojë të jetë në gjendje të ndalojë ekzekutimin e tij në mënyrë të rregullt (end ( )) ose në mënyrë jo të rregullt (abort ( )). Nëse thirret një thirrje sistemore për të ndalur ekzekutimin e programit aktual në mënyrë jo të rregullt, ose nëse programi hasë në ndonjë problem dhe shkakton ndonjë trap gabimi, përmbajtja e memories së programit vendoset në memorie sekondare dhe paraqitet një mesazh gabimi. Përmbajtja e memories së programit shkruhet në një fajll të veçantë në disk dhe mund të kontrollohet nga një **debugger** – një program i sistemit i dizajnuar për të ndihmuar programuesin në gjetjen dhe përmirësimin e gabimeve logjike ose **bugs** – për të përcaktuar rrënjët e problemit. Në rrethana të rregullta ose jo të rregullta, sistemi operativ duhet patjetër të bartë kontrollin te interpretuesi i komandave thirrës. Interpretuesi i komandave lexon komandën e radhës. Në një sistem interaktiv, interpretuesi i komandave thjeshtë vazhdon me komandën e radhës; pritet që shfrytëzuesi do të jepë komandën përkatëse si përgjigje për ndonjë gabim. Në një sistem me GUI, shfrytëzuesit mund t'i paraqitet një mesazh për gabim dhe mund të kërkojë që të ofrojë ndihmë. Disa sisteme mund të lejojnë disa veprime përmirësuese në rast të ndonjë gabimi. Nëse programi vëren ndonjë gabim në hyrjen e tij dhe dëshiron të përfundojë ekzekutimin në mënyrë jo të rregullt, mund të dëshironte të definojë një nivel të gabimit. Gabimet më të mëdha mund të tregohen nga një parametër i gabimit të nivelit më të lartë. Është e mundur që të kombinohet përfundimi i rregullt dhe jo i rregullt duke definuar përfundimin e rregullt si një gabim në nivelin 0. Interpretuesi i komandave ose ndonjë program vijues mund të përdorë këtë nivel të gabimit për të përcaktuar automatikisht veprimin e radhës.

Një proces që është duke ekzekutuar një program mund të dëshirojë të vendosë në DRAM përmes load ( ) dhe ekzekutojë përmes execute ( ) ndonjë program. Kjo mundësi ia lejon interpretuesit të komandave të ekzekutojë një program përmes një komande të shfrytëzuesit ose ndonjë klikimi përmes mausit. Një pyetje e rëndësishme është se ku të kthehet kontrolli kur programi të përfundojë ekzekutimin. Kjo pyetje lidhet me atë se programi ekzistues humbet, ruhet ose lejohet të vazhdojë ekzekutimin konkurent me programin e ri.

Nëse kontrolli i kthehet programit ekzistues kur programi i ri të përfundojë ekzekutimin, duhet patjetër të ruhet përmbajtja e memories së programit ekzistues; kështu, kemi krijuar një

mekanizëm që një program të thërrasë një program tjetër. Nëse të dy programet vazhdojnë në mënyrë konkurente, kemi krijuar një proces të ri për t'u multiprogramuar. Për këtë ekziston një thirrje sistemore (create_process ( )).

Nëse krijojmë një proces të ri ose një bashkësi të proceseve, duhet të jemi në gjendje të kontrollojmë ekzekutimin e tij. Ky kontroll kërkon mundësinë për të përcaktuar dhe rivendosur atributet e një procesi, përfshirë prioritetin e procesit, kohën maksimale të lejueshme për ekzekutimin e tij dhe tjera (get_process_attributes ( ) dhe set_process_attributes ( )). Mund të kemi nevojë që të ndalim procesin që kemi krijuar (terminate_process ( )) nëse vërejmë se është jo i duhuri ose nuk nevojitet më.

Pasi të kemi krijuar disa procese, mund të kemi nevojë që të presim ekzekutimin e tyre. Mund të kemi nevojë të presim që të kalojë një kohë (wait_time ( )). Me gjasë, mund të kemi nejovë të presim për ndonjë ngjarje specifike (wait_event ( )). Proceset pastaj duhet të sinjalizojnë që ngjarja është paraqitur (signal_event ( )).

Shumë shpesh, dy ose më shumë procese mund të përdorin të dhënat e njëjta. Për të siguruar integritetin e të dhënave të tilla, sistemet operative shpesh ofrojnë thirrje sistemore që lejojnë një proces të **mbyllë (lock)** të dhënat e përbashkëta (që përdoren nga proceset). Në atë rast, asnjë proces tjetër nuk mund të qaset në të dhëna derisa të lirohen të dhënat (nga procesi që i ka mbyllur ato). Zakonisht, thirrjet sistemore si këto përfshijnë thirrjet acquire_lock ( ) dhe release_lock ( ).

Ekzistojnë shumë mundësi dhe mënyra të kontrollit të procesit ashtu që në vazhdim do të përdorim dy shembuj – njëri që përfshinë një sistem që mund të ekzekutojë vetëm një punë dhe njëri një sistem që mund të ekzekutojë më shumë se një punë – për të sqaruar këto koncepte. Arduino është një platformë e thjeshtë harduerike që përbëhet nga një mikrokontrollues bashkë me disa sensorë hyrës që reagojnë ndaj ngjarjeve të ndryshme, si ndryshimet në ndriçim, temperaturë dhe shtypje të ajrit. Për të shkruar një program për Arduino, së pari programin e shkruajmë përmes një kompjuteri dhe pastaj vendosim programin e kompajluar (të njohur si sketch) nga kompjuteri në memorien flash të Arduinos përmes një lidhjeje me USB. Platforma standarde Arduino nuk ofron një sistem operativ; në vend të tij, një program i vogël i njohur si **boot loader** e vendos sketch-in në një pjesë specifike në memorien e Arduinos (Figura 2.9).

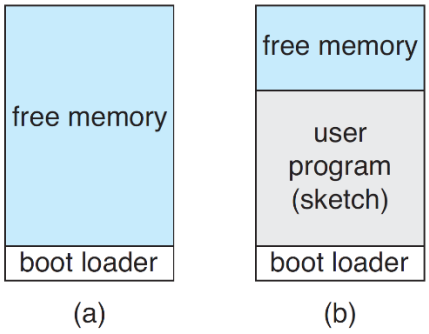


Figura 2.9 Ekzekutimi në Arduino. (a) Gjatë startimit të sistemit. (b) Ekzekutimi i sketch.

Kur sketch të jetë vendosur në memorie, fillon ekzekutimin, duke pritur për paraqitjen e ngjarjeve për të cilat është programuar të reagojë. Për shembull, nëse sensori i temperaturës në Arduino vëren se temperatura e ka kaluar një vlerë të caktuar (threshold), sketch e detyron Arduinon të startojë motorrin e një ftohësi (të startojë ftohësin). Arduino është një sistem që mund të ekzekutojë vetëm një punë në një moment të caktuar, meqë vetëm një sketch mund të jetë prezent në memorie në një moment të caktuar; nëse vendoset ndonjë sketch tjetër, ai zëvendëson sketchi-n ekzistues në memorie. Për më tepër, Arduino nuk ofron ndonjë interfejs për shfrytëzuesin përveç sensorëve harduerik për hyrje.

FreeBSD (që rrjedhë nga Berkley UNIX) është një shembull i një sistemi multitasking. Kur një shfrytëzues logohet në sistem, shell-i që ka përzgjedhë shfrytëzuesi për përdorim ekzekutohet, duke pritur për komanda dhe bërë ekzekutimin e programeve që kërkon shfrytëzuesi. Megjithatë, meqë FreeBSD është një sistem multitasking, interpretuesi i komandave mund të vazhdojë ekzekutimin përderisa një program tjetër ekzekutohet (Figura 2.10).

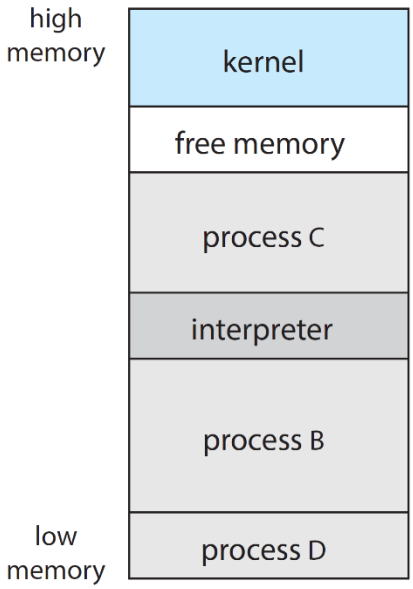


Figura 2.10 FreeBSD duke ekzekutuar shumë programe

Për të filluar një proces të ri, shell ekzekuton thirrjen sistemore fork ( ). Pastaj, programi i përzgjedhur vendoset në memorie përmes thirrjes sistemore exec ( ) dhe ekzekutohet programi. Varësisht se si është dhënë komanda, shell pret që të përfundojë procesi ose e ekzekuton në "prapavijë". Në rastin e dytë, shell pret që të futet një komandë tjetër. Kur një proces është duke u ekzekutuar në prapavijë, nuk mund të pranojë hyrje direkt prej tastierës, sepse shell është duke përdorë tastierën. Hyrje/dalja realizohet përmes fajllave ose përmes një interfejsi me GUI. Ndërkohë, shfrytëzuesi është i lirë të kërkojë nga shell që të ekzekutojë programe tjera, të monitorojë progresin e procesit që është duke u ekzekutuar, të ndryshojë prioritetin e programit e të tjera. Kur procesi të përfundojë, ai e ekzekuton një thirrje sistemore exit ( ) për të përfunduar, duke ia kthyer procesit thirrës kodin 0 të statusit ose një kod gabimi të ndryshëm nga zero. Ky kod i statusit ose gabimit është në dispozicion të shell-it ose të programeve tjerë.

2.3.3.2 Menaxhimi i fajllave

Duhet të jemi në gjendje të krijojmë fajlla përmes create ( ) dhe fshijmë përmes delete ( ). Secila nga këto thirrje sistemore kanë nevojë për emrin e fajllit dhe ndoshta për disa atribute të fajllit. Pas krijimit të fajllit, kemi nevojë të hapim fajllin përmes open ( ) dhe të përdorim atë. Fajllin mund të lexojmë përmes read ( ), shkruajmë përmes write ( ) ose lëvizim brenda fajllit përmes reposition ( ) (për shembull, kalimi deri në fund të fajllit). Në fund mund të mbyllim fajllin përmes close ( ) për të treguar se kemi përfunduar punën me fajllin.

Mund të kemi nevojë për të njëjtat operacione (veprime) për direktoriume nëse kemi një strukturë të direktoriumeve për organizimin e fajllave në sistemin e fajllave. Përveç kësaj, për fajlla ose direktoriume, mund të kemi nevojë të përcaktojmë vlerat e atributeve të ndryshme dhe vendosjen e vlerave të tyre sipas nevojës. Atributet e fajllit përfshijnë emrin e fajllit, llojin (tipin) e fajllit, kodet e mbrojtjes e të tjera. Për këtë funksion nevojiten së paku dy thirrje sistemore: get_file_attributes ( ) dhe set_file_attributes ( ). Disa sisteme operative ofrojnë më shumë thirrje sistemore, si thirrjet për bartjen e fajllit përmes move ( ) dhe kopjimin e fajllit përmes copy ( ). Disa sisteme tjera mund të ofrojnë një API që i kryen ato operacione duke përdorë kod dhe thirrje sistemore tjera, e disa të tjerë ofrojnë programe të sistemit për të kryer ato punë. Nëse programet e sistemit mund të thirren nga programet tjera, atëherë secili prej tyre mund të konsiderohet si API nga programet tjera të sistemit.

2.3.3.3. Menaxhimi i pajisjes

Një proces mund të ketë nevojë për disa resurse për t'u ekzekutuar – memoria DRAM, disqet, qasja në fajlla e të tjera. Nëse resurset janë në dispozicion, ato mund të jepen në shfrytëzim dhe kontrolli mund t'i kthehet procesit të shfrytëzuesit. Përndryshe, procesi duhet të presë që resurset e mjaftueshme të jenë në dispozicion.

Resurset e ndryshme që kontrollohen nga sistemi operativ mund të mendohen si pajisje. Disa prej këtyre pajisjeve janë pajisje fizike (për shembull, disqet), kurse tjerat mund të mendohen si pajisje virtuale (për shembull, fajllat). Një sistem me shumë shfrytëzues mund të kërkojë që shfrytëzuesi së pari të marrë në shfrytëzim një pajisje përmes request ( ), për t'u siguruar se ka përdorim ekskluziv të saj. Pasi të përfundojë punën me pajisje, shfrytëzuesi e liron atë përmes release ( ). Këto funksione janë të ngjashme me thirrjet sistemore open ( ) dhe close ( ) për fajlla. Disa sisteme operative lejojnë qasje jo të menaxhuar në pajisje. Dëmi që mund të shkaktojë kjo është mundësia që pajisjet të mbesin te vetëm një shfrytëzues dhe paraqitja e situatës deadlock.

Kur pajisja të jetë marrë në shfrytëzim, mund të lexojmë përmes read ( ), shkruajmë përmes write ( ) dhe lëvizim në pajisje përmes reposition ( ), njëjtë sikur me fajlla. Në të vërtetë, ngjashmëria në mes të pajisjeve H/D dhe fajllave është aq e madhe sa që shumë sisteme operative, përfishirë UNIX, i bashkojnë te dy në një strukturë të kombinuar fajll-pajisje. Në këtë rast, një bashkësi e njëjtë e thirrjeve sistemore përdoret për fajlla dhe pajisje. Në disa raste, pajisjet H/D identifikohen me emra të veçantë të fajllave, vendin e direktoriumit ose atributet e fajllit.

Interfejsi i shfrytëzuesit poashtu mund t'i paraqesë njëjtë fajllat dhe pajisjet, edhepse thirrjet sistemore të tyre janë të ndryshme. Ky është një shembull tjetër i shumë vendimeve të dizajnit që merren për ndërtimin e një sistemi operativ dhe të interfejsit të shfrytëzuesit.

2.3.3.4 Mirëmbajtja e informacionit

Shumë thirrje sistemore ekzistojnë me qëllim të bartjes së informacionit në mes të programit të shfrytëzuesit dhe sistemit operativ. Për shembull, shumica e sistemeve kanë një thirrje sistemore për të lexuar kohën aktuale përmes time ( ) dhe datën përmes date ( ). Disa thirrje sistemore tjera mund të ofrojnë informata rreth sistemit, si numrin e versionit të sistemit operativ, sasinë e memories së lirë ose hapësirën në disk e të tjera.

Një bashkësi tjetër e thirrjeve sistemore janë të dobishme për të bërë debugging të një programi. Shumë sisteme ofrojnë thirrje sistemore për të shkruar përmbajtjen e programit prej memories DRAM në disk përmes dump ( ). Kjo është e rëndësishme për debugging. Programi strace, që është në sistemet Linux, paraqet listën me secilën thirrje sistemore kur janë duke u ekzekutuar. Edhe procesorët ofrojnë një mënyrë të punës, e njohur si **me një hap (single step)**, në të cilën ekzekutohet një trap nga procesori pas çdo instruksioni. Zakonisht trap trajtohet nga një debugger.

Shumë sisteme operative ofrojnë një profil kohor për një program për të treguar sa kohë ekzekutohet programi në një vend ose bashkësi të vendeve të caktuara. Një profil kohor kërkon një mundësi për gjurmim ose ndërprerje të rregullta prej tajmerit. Në çdo ndërprerje prej tajmerit, vlera e regjistrit PC ruhet për të ditur vlerën që është në regjistrin PC. Me një numër të mjaftueshëm të ndërprerjeve prej tajmerit, mund të merret një fotografi statistikore e kohës së procesorit të kaluar në pjesë të ndryshme të programit.

Për më tepër, sistemi operativ mbanë informata për të gjitha proceset e tij dhe thirrjet sistemore përdoren për të lexuar atë informacion. Në përgjithësi, thirrjet përdoren për të lexuar dhe vendosur informatat e procesit (get_process_attributes ( ) dhe set_process_attributes ( )).

2.3.3.5 Komunikimi

Janë dy mënyra të zakonshme të komunikimit në mes të proceseve: modeli me shkëmbim të mesazheve dhe modeli me memorie të përbashkët. Te **modeli me shkëmbim të mesazheve**, proceset komunikuese shkëmbejnë mesazhet me njëri-tjetrin për të bartur informacionin. Mesazhet mund të shkëmbehen direkt ose me anë të një kutie postare të përbashkët. Para se të ndodhë komunikimi, duhet patjetjër të hapet një lidhje komunikimi. Duhet patjetër të dihet emri i palës tjetër komunikuese, qoftë ai një proces në sistemin e njëjtë ose një proces në një kompjuter tjetër të lidhur me një rrjetë komunikuese. Çdo kompjuter në një rrjetë ka **emrin e hostit** me të cilin dallohet. Një host ka poashtu edhe një numër identifikues brenda rrjetës, një IP adresë. Ngjashëm, çdo proces ka **emrin e procesit** që përkthehet në një identifikues me të cilin sistemi operativ mund t'i referohet procesit. Ky përkthim bëhet përmes thirrjeve sistemore get_hostid ( ) dhe get_processid ( ). Identifikuesit pastaj barten te thirrjet për qëllime të përgjithshme open ( ) dhe close ( ) që ofrohen nga sistemi i fajllave ose te thirrjet sistemore specifike open_connection ( ) dhe close_connection ( ), varësisht prej modelit të komunikimit në sistem. Procesi pranues duhet patjetër të jepë pëlqimin për komunikim përmes një thirrjeje accept_connection ( ).

Shumica e proceseve që do të pranojnë kërkesat për lidhje komunikimi janë programe për qëllime të veçanta të njohur si **daemons**, që janë programe të sistemit që ofrohen për atë qëllim. Ato ekzekutojnë një thirrje wait_for_connection ( ) dhe aktivizohen kur të realizohet lidhja. Burimi i komunikimit, i njohur si **klienti** dhe daemon pranues, i njohur si **serveri**, shkëmbejnë mesazhet përmes thirrjeve sistemore read_message ( ) dhe write_message ( ). Thirrja close_connection ( ) mbyllë lidhjen e komunikimit.

Në **modelin me memorie të përbashkët**, proceset përdorin thirrjet shared_memory_create ( ) dhe shared_memory_attach ( ) për të krijuar dhe marrë qasje në pjesët e memories të proceseve tjera. Të kujtojmë se sistemi operativ tenton parandalimin e një procesi që t'i qaset memories së një procesi tjetër. Memoria e përbashkët kërkon që të dy ose më shumë proceset të heqin këtë kufizim. Proceset mund të shkëmbejnë informacion duke lexuar dhe shkruar të dhëna në pjesët e përbashkëta të memories. Forma e të dhënave përcaktohet nga proceset dhe nuk është nën kontrollin e sistemit operativ. Proceset janë poashtu përgjegjëse për t'u siguruar se ato nuk janë duke shkruar në të njëjtën kohë në të njëjtin vend.

Të dy këto modele janë të zakonshme në sisteme operative dhe shumica e sistemeve implementojnë të dy këto modele. Shkëmbimi i mesazheve është i dobishëm për shkëmbimin e sasive të vogla të të dhënave sepse nuk është e nevojshme të shmanget ndonjë konflikt. Ky model implementohet më lehtë se modeli me memorie të përbashkët për komunikime në mes të kompjuterëve. Modeli me memorie të përbashkët mundëson shpejtësi maksimale të bartjes dhe lehtësi komunikimi, meqë mund të bëhet në nivelin e shpejtësisë së bartjes të memories kur ndodhë në një kompjuter. Megjithatë, ekzistojnë disa probleme sa i përket mbrojtjes dhe sinkronizimit në mes të proceseve që përdorin memorien e njëjtë.

2.3.3.6 Mbrojtja

Mbrojtja ofron një mekanizëm për kontrollimin e qasjes në resurset që ofrohen nga një sistem kompjuterik. Historikisht, mbrojtja ishte një shqetësim vetëm në sistemet e multiprogramuara me shumë shfrytëzues. Megjithatë, me shpikjen e rrjetës dhe Internetit, të gjitha sistemet kompjuterike, që nga serverët e deri te pajisjet mobile, duhet patjetër të jenë të shqetësuara për mbrojtjen.

Zakonisht, thirrjet sistemore që ofrojnë mbrojtje përfshijnë set_permission ( ) dhe get_permission ( ), të cilat punojnë me konfigurimet në lidhje me të drejtat e përdorimit të resurseve si fajllat dhe disqet. Thirrjet sistemore allow_user ( ) dhe deny_user ( ) përcaktojnë nëse ndonjë shfrytëzuesi të caktuar mund ose nuk mund t'i lejohet qasja në resurse të caktuara.

2.4 Shërbimet e sistemit

Një aspekt tjetër i një sistemi modern është bashkësia e shërbimeve të sistemit. Të kujtojmë Figura 1.1, në të cilën ishte paraqitur struktura hierarkike e kompjuterit. Në nivelin më të ulët është hardueri. Pastaj vjen sistemi operativ, shërbimet e sistemit dhe në fund programet aplikative. **Shërbimet e sistemit**, të njohura edhe si **system utilities**, ofrojnë një ambeint të përshtatshëm për zhvillimin dhe ekzekutimin e programit. Disa prej tyre janë thjeshtë interfejs i

shfrytëzuesit për thirrjet sistemore. Tjerat janë dukshëm më të komplikuara. Ato mund të ndahen në këto grupe:

- **Menaxhimi i fajllave**. Këto programe krijojnë, fshijnë, kopjojnë, riemërtojnë, printojnë, listojnë dhe në përgjithësi qasen dhe punojnë me fajlla dhe direktoriume.
- **Informacioni i statusit**. Disa programe kërkojnë nga sistemi datën, kohën, hapësirën e lirë të memories DRAM ose diskut, numrin e shfrytëzuesve ose informata të ngjashme të statusit. Tjerët janë më të komplikuar, duke ofruar informata të detajuara, ruajtje të informatave dhe informacion për debugging. Zakonisht, këto programe formatojnë dhe paraqesin daljen në një ekran ose në pajisje tjera dalëse ose fajlla apo e shfaqin atë në një dritare në GUI. Disa sisteme ofrojnë edhe një **registry (regjistër)**, që përdoret për të ruajtur dhe lexuar informacion të konfigurimit.
- **Ndryshimi i fajllave**. Mund të ofrohen disa editorë të tekstit për të krijuar dhe ndryshuar përmbajtjen e fajllave të ruajtur në disk ose në pajisje tjera për ruajtje. Mund të jenë edhe disa komanda speciale për të kërkuar brenda përmbajtjes së fajllit ose për të bërë ndryshime në tekst.
- **Përkrahja për gjuhë programuese**. Kompajlerë, asemblerë, debuggers dhe interpretues për gjuhë programuese (si C, C++, Java dhe Python) shpesh ofrohen nga sistemi operativ ose mund të shkarkohen e instalohen ndaras.
- **Vendosja e programit në memorie dhe ekzekutimi**. Pas asemblimit ose kompajlimit të një programi, programi duhet patjetër të vendoset në memorien DRAM për ekzekutim. Sistemi mund të ofrojë edhe programe si: absolute loaders, relocatable loaders, linkage editors dhe overlay loaders. Nevojiten edhe sisteme për debugging për gjuhë programimi të larta ose për gjuhën e makinës.
- **Komunikimet**. Këto programe ofrojnë mekanizmin për krijimin e lidhjeve virtuale në mes proceseve, shfrytëzuesve dhe sistemeve kompjuterike. Ato mundësojnë shfrytëzuesit të dërgojnë mesazhe në ekranet e njëri-tjetrit, të shfletojnë ueb faqe, të dërgojnë mesazhe e-mail, të logohen në distancë ose të bartin fajlla prej një kompjuteri në një tjetër.
- **Shërbimet në prapavijë**. Të gjitha sistemet për qëllime të përgjithshme kanë metoda për të startuar disa procese të programeve të sistemit gjatë kohës të lëshimit në punë të sistemit (boot time). Disa prej këtyre proceseve ndalin ekzekutimin pas përfundimit të punëve të tyre, përderisa disa të tjera vazhdojnë ekzekutimin deri në ndaljen e punës në kompjuter. Programet e sistemit që ekzekutohen vazhdimisht njihen si **shërbime**, **nënsisteme**, ose daemons. Një shembull është daemon për rrjetën i diskutuar te pjesa 2.3.3.5 e materialit. Në atë shembull, sistemi kishte nevojë për një shërbim për të dëgjuar (pranuar) lidhjet në rrjetë për të lidhur ato kërkesa me proceset e duhura. Shembuj tjerë përfshijnë proceset për planifikim që startojnë procese sipas një plani specifik, shërbimeve për monitorim të gabimeve në sistem dhe serverëve për printim. Sistemet përdorin disa programe daemon. Për më tepër, sistemet operative që ekzekutojnë punë të rëndësishme në mënyrën e shfrytëzuesit në vend të mënyrës së kernelit mund të përdorin programet daemon për të ekzekutuar këto punë.

Krahas programeve të sistemit, shumica e sistemeve operative kanë programe që janë të dobishme për zgjidhjen e problemeve ose kryerjen e punëve të zakonshme. Këto **aplikacione të shfrytëzuesit** përfshijnë ueb shfletuesit, programet për përpunimin dhe formatimin e

teksteve, programet për llogaritje, sistemet e bazës së të dhënave, kompajlerët, pakotë për vizatim, analiza statistikore dhe lojërat.

Shumica e shfrytëzuesve e shohin sistemin operativ si grup të programeve aplikative dhe programeve të sistemit, e jo si thirrje sistemore. Le të marrim një shfrytëzues të kompjuterit. Nëse në kompjuterin e shfrytëzuesit është sistemi operativ macOS, shfrytëzuesi sheh interfejsin GUI përmes mausit dhe dritareve. Ndryshe, ose në njërën prej dritareve, shfrytëzuesi mund të ketë një shell UNIX me komanda. Te dy përdorin të njëjtat thirrje sistemore, por thirrjet sistemore duken ndryshe dhe punojnë në mënyra të ndryshme. Për të bërë edhe më konfuze atë se si e sheh sistemin shfrytëzuesi, le të marrin rastin kur shfrytëzuesi përmes dual booting kyçet në Windows nga macOS. Në këtë rast i njëjti shfrytëzues në të njëjtin harduer ka dy interfejse krejtësisht të ndryshme dhe dy grupe të aplikacioneve që përdorin të njëjtat resurse fizike. Atëherë, në të njëjtin harduer, një shfrytëzues mund të përballet me disa interfejse të shfrytëzuesit në mënyrë sekuenciale ose konkurente.

2.5 Linkers dhe loaders

Zakonisht, një program në formë binare të ekzekutueshme ndodhet në një fajll në disk – për shembull, a.out ose prog.exe. Për t'u ekzekutuar në CPU, programi duhet patjetër të vendoset në DRAM dhe të vendoset në kontekstin e një procesi. Në këtë pjesë, do të përshkruajmë hapat në këtë procedurë, që nga kompajlimi i një programi e deri të vendosja e tij në memorien DRAM, ku prej aty mund të ekzekutohet në ndonjë bërthamë të CPU-së. Këta hapa janë paraqitur në Figura 2.11.

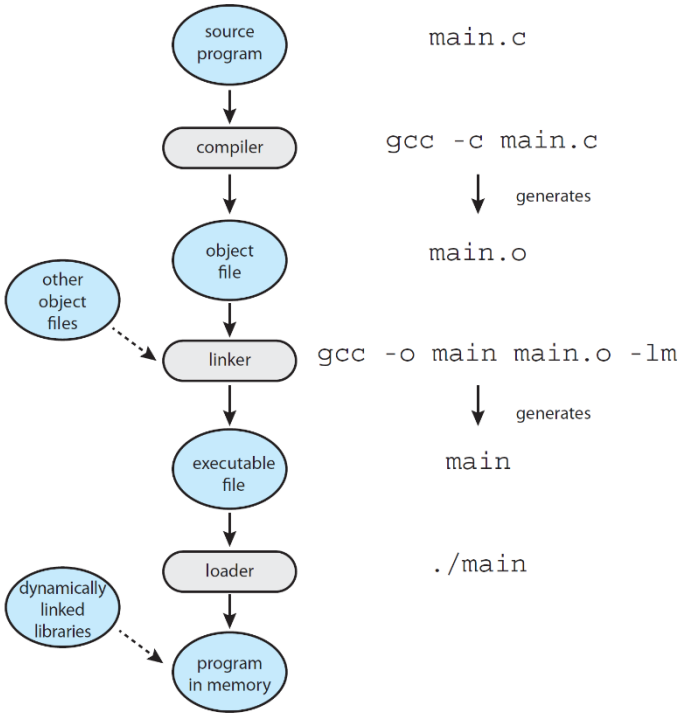


Figura 2.11 Roli i linker dhe loader

Fajllat me kod burimor kompajlohen në fajlla objekt (binarë) që janë të dizajnuar të vendosen në cilëndo pjesë të memories DRAM, një format që njihet si fajll objekt i zhvendosshëm (relocatable object file). Në vazhdim, **linker** i kombinon këta fajlla objekt të zhvendosshëm në një fajll të vetëm binar të ekzekutueshëm. Gjatë fazës së kombinimit (lidhjes), fajlla tjerë objekt ose librari mund të përfshihen, si libraria standarde e gjuhës C ose libraria me funksione matematikore (përmes opcionit -lm).

Loader përdoret për të vendosur në memorie fajllin binar të ekzekutueshëm, prej aty mund të ekzekutohet në ndonjë bërthamë të CPU-së. Një veprim që lidhet me linking dhe loading është **zhvendosja (relocation)**, që jepë adresat përfundimtare për pjesët e programit dhe rregullon kodin dhe të dhënat në program që të lidhen me ato adresa, për shembull, kodi mund të thërrasë një librari me funksione dhe qaset në variablat e tij gjatë ekzekutimit. Në Figura 2.11, mund të shohim se për të ekzekutuar loader, e tërë ajo që duhet të bëjmë është të japim emrin e fajllit të ekzekutueshëm përmes interfejsit me komanda. Kur të jepet emri i programit përmes komandave në sistemet UNIX – për shembull, ./main – së pari shell krijon një proces për të ekzekutuar programin përmes thirrjes sistemore fork ( ). Në vazhdim, shell thërret loader përmes thirrjes sistemore exec ( ), duke ia bartur thirrjes exec ( ) emrin e fajllit të ekzekutueshëm. Loader vendosë atë program në memorien DRAM duke përdorë hapësirën e adresave të procesit të ri të krijuar. (Kur përdoret një interfejs GUI, klikimi dy herë mbi ikonën e fajllit të ekzekutueshëm thërret loader duke përdorë mekanizmin e njëjtë).

Procedura që përshkruam deri më tani supozon që të gjitha libraritë lidhen në një fajll të ekzekutueshëm dhe vendosen në memorie. Në të vërtetë, shumica e sistemeve lejojnë që një program të lidhet në mënyrë dinamike me libraritë gjatë vendosjes së programit në DRAM. Për shembull, Windows, përkrahë **dynamically linked libraries (DLL)**. Përparësi e kësaj qasjeje është se shmangë nevojën për linking dhe loading të librarive të cilat mund të mos përdoren fare në një fajll të ekzekutueshëm. Për dallim, libraria lidhet nëse duhet dhe vendoset në memorie nëse nevojitet gjatë ekzekutimit të programit. Për shembull, në Figura 2.11, libraria me funksione matematikore nuk lidhet në fajllin e ekzekutueshëm main. Në vend të lidhjes, linker vendosë informata të zhvendosjes që ia mundësojnë atij të lidhet në mënyrë dinamike dhe të vendoset në memorie kur të vendoset programi. Shumë procese mund të përdorin të njëjtat librari që mund të lidhen në mënyrë dinamike, duke mundësuar kursim të madh të përdorimit të memories.

Fajllat objekt dhe të ekzekutueshëm zakonisht kanë një format standard që përfshinë kodin e kompajluar të makinës dhe një tabelë të simboleve që përmban të dhëna (metadata) për funksionet dhe variablat që përdoren në program. Për sistemet UNIX dhe Linux, ky format standard është i njohur si ELF (**executable and linkable format)**. Janë dy formate ELF të ndara për fajllat e zhvendosshëm dhe ata të ekzekutueshëm. Një informatë brenda fajllit ELF për fajllat e ekzekutueshëm është pjesa e fillimit të ekzekutimit të programit (**entry point)**, që përmban adresën e instruksionit të parë që duhet të ekzekutohet kur të fillojë ekzekutimi i programit. Sistemet Windows përdorin formatin **Portable Executable** (PE) dhe sistemet macOS përdorin formatin **Mach-O**.

2.6 Pse programet shkruhen për sisteme operative të caktuara?

Në thelb, programet që kompajlohen në një sistem operativ nuk janë të ekzekutueshme në sisteme operative tjera. Nëse do të ishin të ekzekutueshme, bota do të ishte një vend më i mirë dhe zgjedhja jonë se cilin sistem operativ të përdorim do të varej nga mundësitë dhe karakteristikat e jo nga ajo se cilat aplikacione janë në dispozicion.

Bazuar në atë që kemi trajtuar më herët, tash mund të shohim një pjesë të problemit – çdo sistem operativ ofron një bashkësi unike të thirrjeve sistemore. Thirrjet sistemore janë pjesë e bashkësisë së shërbimeve të ofruara nga sistemet operative për përdorim nga aplikacionet. Edhe po të ishin thirrjet sistemore uniforme, pengesa tjera do të mund të na bënin të vështirë ekzekutimin në sisteme operative tjera. Nëse keni përdorë sisteme operative të ndryshme mund të keni përdorë disa aplikacione të njëjta në ato sisteme operative. Si është e mundur një gjë e tillë?

Një aplikacion mund të shkruhet për t'u ekzekutuar në sisteme operative të ndryshme në një njërën prej tre mënyrave:

1. Aplikacioni mund të shkruhet përmes një gjuhe programuese të interpretuar (Python ose Ruby) që ka nga një interpretues për sisteme operative të ndryshme. Interpretuesi lexon secilin rresht në programin burimor, ekzekuton instruksionet përkatëse nga bashkësia e instruksioneve të procesorit dhe thërret thirrjet sistemore të sistemit operativ. Kjo mund të ketë ndikim në performancë dhe interpretuesi ofron vetëm pjesë të karakteristikave të sistemit operativ, me mundësinë e kufizimit të karakteristikave të aplikacioneve.
2. Aplikacioni mund të shkruhet përmes një gjuhe që përfshinë një makinë virtuale e cila përmban programin që është duke u ekzekutuar. Makina virtuale është pjesë e RTE-së së gjuhës. Një shembull është Java. Java ka një RTE që përfshinë loader, verifikues të formatit byte-code dhe pjesë tjera të cilat e vendosin aplikacionin e shkruar në Java në makinën virtuale të Java-s. Kjo RTE është bartur (ported) ose zhvilluar për shumë sisteme operative, që nga kompjuterët mainframe e deri te telefonët e mençur dhe në teori çdo aplikacion në Java mund të ekzekutohet në RTE kur të jetë e lirë. Sistemet si këto kanë mangësitë e tyre ngjashëm me ato të interpretuesve.
3. Zhvilluesi i aplikacioneve mund të përdorë një gjuhë standarde ose API në të cilën kompajleri krijon fajllat binarë në një gjuhë specifike për harduerin dhe sistemin operativ të caktuar. Aplikacioni duhet patjetër të bartet në secilin sistem operativ në të cilin do të ekzekutohet. Ajo bartje mund të marrë shumë kohë dhe duhet të bëhet patjetër për secilin version të ri të aplikacionit, me testim dhe debugging. Mbase shembulli më i njohur është POSIX API dhe standardet e tij për mirëmbajtjen e përputhshmërisë së kodit burimor në mes të varianteve të ndryshme të sistemeve operative të ngjashme me UNIX.

Në teori, duket se këto tre qasje ofrojnë zgjidhje të thjeshta për zhvillimin e aplikacioneve që mund të ekzekutohen në sisteme operative të ndryshme. Megjithatë, mungesa e mundësisë për bartje të aplikacionit ka disa arsye, të gjitha ato e bëjnë sfiduese zhvillimin e aplikacioneve për platforma (sisteme) të ndryshme. Në nivelin e aplikacionit, libraritë që ofrohen nga sistemi

operativ kanë API për të ofruar karakteristika si interfejsi GUI dhe një aplikacion i dizajnuar për të thirrë një pjesë të API-ve (le të themi, ato që janë në iOS në Apple iPhone) nuk mund të ekzekutohet në një sistem operativ që nuk i ofron ato API (si për shembull Android). Ekzistojnë edhe sfida tjera në nivelet më të ulëta në sistem, duke përfshirë këto:

- Çdo sistem operativ e ka një format binar për aplikacionet që përcakton shtrirjen e pjesës header, instruksioneve dhe variablave. Ato pjesë duhet të jenë në vende të caktuara në struktura të caktuara brenda një fajlli të ekzekutueshëm ashtu që sistemi operativ mund të hapë fajllin dhe të vendosë në memorie aplikacionin për ekzekutimin e duhur.
- CPU-të kanë bashkësi të ndryshme të instruksioneve dhe vetëm ato aplikacione që kanë instruksionet e duhura (të lejuara) mund të ekzekutohen.
- Sistemet operative ofrojnë thirrje sistemore që mundësojnë aplikacionet të kërkojnë ndihmë për punë të ndryshme si krijimi i fajllave dhe hapja e një lidhjeje në rrjetë. Ato thirrje sistemore dallojnë shumë në shumë aspekte në sisteme operative të ndryshme, përfshirë operandët e caktuar dhe mënyrën e radhitjes së operandëve që përdoren, si një aplikacion i thërret thirrjet sistemore, numërimi i tyre dhe numri, kuptimet e tyre dhe kthimi i rezultateve prej tyre.

Janë disa qasje që kanë ndihmuar në adresimin, edhepse jo për të zgjidhur krejt problemin, e dallimeve në arkitekturë. Për shembull, Linux – dhe pothuajse çdo sistem UNIX – kanë adaptuar formatin ELF për fajllat e ekzekutueshëm. Edhepse ELF ofron një standard të vetëm përgjatë sistemeve Linux dhe UNIX, ELF nuk është i lidhur (nuk varet) me ndonjë arkitekturë specifike të kompjuterit, ashtu që nuk garanton që një fajll i ekzekutueshëm do të ekzekutohet përgjatë platformave të ndryshme harduerike.

API-të specifikojnë funksione të caktuara në nivelin e aplikacionit. Në nivelin e arkitekturës, përdoret një **application binary interface (ABI)** për të përcaktuar se si pjesë të ndryshme të kodit binar mund të përdoren për një sistem të caktuar operativ në një arkitekturë të caktuar. Një ABI specifikon detaje të nivelit të ulët, përfshirë gjerësinë e adresës, mënyrat e bartjes së parametrave te thirrjet sistemore, organizimin e stekut për ekzekutim, formatin binar të librarive të sistemit, madhësinë e tipeve të të dhënave etj. Zakonisht, një ABI specifikohet për një arkitekturë të caktuar (për shembull, ekziston një ABI për procesorin ARMv8). Kështu, një ABI është ekuivalenti në nivel të arkitekturës i një API. Nëse një fajll binar i ekzekutueshëm është kompajluar dhe lidhur sipas një ABI, duhet të jetë në gjendje të ekzekutohet në sisteme të ndryshme që përkrahin atë ABI. Megjithatë, meqë një ABI i caktuar është përcaktuar për një sistem operativ të caktuar që ekzekutohet në një arkitekturë, ABI-të ndihmojnë pak për përputhshmëri përgjatë platformave.

Si përmbledhje, të gjitha këto ndryshime nënkuptojnë që përveç nëse një interpretues, RTE ose fajll binar i ekzekutueshëm shkruhet dhe kompajlohet në një sistem operativ të caktuar në një CPU të caktuar (në Intel x86 ose ARMv8), aplikacioni nuk do të mund të ekzekutohet. Keni parasysh punën që duhet kryer për një program si shfletuesi Firefox për ekzekutim në Windows, macOS, disa versione të Linux, iOS dhe Android, në disa arkitektura të ndryshme të CPU-së.

2.7 Dizajnimi dhe implementimi i sistemit operativ

Në këtë pjesë, do të trajtojmë problemet me të cilat përballemi gjatë dizajnimit dhe implementimit (realizimit) të një sistemi operativ. Nuk ka zgjidhje të plota për këto probleme, por janë disa qasje që janë treguar të suksesshme.

2.7.1 Qëllimet e dizajnit

Problemi i parë për dizajnimin e një sistemi është përcaktimi i qëllimeve dhe specifikimeve. Në nivelin më të lartë, dizajni i sistemit do të ndikohet nga zgjedhja e harduerit dhe llojit të sistemit: desktop/laptop tradicional, mobil, i shpërndarë ose në kohë reale.

Përtej këtij nivelit më të lartë të dizajnit, kërkesat mund të jenë më të vështira për t'u specifikuar. Megjithatë, kërkesat mund të ndahen në dy grupe bazë: **qëllimet e shfrytëzuesit** dhe **qëllimet e sistemit.**

Shfrytëzuesit duan disa veçori të caktuara në sistem. Sistemi duhet të jetë i përshtatshëm për përdorim, i lehtë për t'u mësuar, i besueshëm, i sigurtë dhe i shpejtë. Sigurisht, këto specifikime nuk janë veçanërisht të dobishme për dizajnimin e sistemit, meqë nuk ka një pajtim të përgjithshëm se si të arrihen ato.

Një grup i njëjtë i kërkesave mund të përcaktohet nga zhvilluesit të cilët duhet patjetër të dizajnojnë, krijojnë, mirëmbajnë dhe punojnë me sistemin. Sistemi duhet të jetë i lehtë për t'u dizajnuar, implementuar dhe mirëmbajtur; dhe duhet të jetë fleksibil, i besueshëm, pa gabime dhe efikas. Këto kërkesa janë të paqarta dhe mund të interpretohen në mënyra të ndryshme.

Thjeshtë, nuk ka një zgjidhje unike për problemin e përcaktimit të kërkesave për një sistem operativ. Shumë sisteme të ndryshme që ekzistojnë tregojnë se kërkesat e ndryshme mund të çojnë kah një numër i madh i zgjidhjeve për ambiente të ndryshme. Për shembull, kërkesat për Wind River VxWorks, një sistem operativ në kohë reale për sistemet embedded, duhet patjetër të kenë qenë thelbësisht të ndryshme prej atyre për Windows Server, një sistem operativ i madh i dizajnuar për aplikacionet që përdoren në ndërmarrje.

Specifikimi dhe dizajnimi i një sistemi operativ është një punë mjaft kreative. Edhepse nuk ka rregulla të përgjithshme se si duhet bërë këtë punë, janë zhvilluar disa parime të përgjithshme në fushën e **inxhinierisë softuerike** dhe në vazhdim do të trajtojmë disa prej atyre parimeve.

2.7.2 Mekanizmat dhe teknikat (politikat)

Një parim me rëndësi është ndarja e **teknikës** prej mekanizmit. Mekanizmat përcaktojnë *si* të bëhet ndonjë punë; teknikat (politikat) përcaktojnë *çka* duhet bërë. Për shembull, konstrukti i tajmerit (shikoni në Figura 1.4.3) është një mekanizëm për të siguruar mbrojtjen e CPU-së, por vendimi se për sa gjatë duhet të vendoset vlera e tajmerit është një vendim i teknikës.

Ndarja e teknikës prej mekanizmit është me rëndësi për fleksibilitet. Teknikat mund të ndryshojnë përgjatë vendeve ose me kalimin e kohës. Në rastin më të keq, çdo ndryshim në

teknikë do të kërkonte një ndryshim në mekanizmin përkatës. Preferohet një mekanizëm i përgjithshëm i mjaftueshëm për të punuar në teknika të ndryshme. Një ndryshim në teknikë do të kërkonte ripërcaktimin e vetëm disa parametrave të sistemit. Për shembull, le të marrim mekanizmin e dhënies së prioritetit për disa programe ndaj programeve tjera. Nëse mekanizmi është ndarë mirë nga teknika, mund të përdoret për të përkrahur një vendim të teknikës që programet që bëjnë shumë hyrje/dalje duhet të kenë përparësi ndaj programeve që përdorin shumë CPU-në ose të përkrahin politikën e kundërt.

Sistemet operative të bazuara në mikrokernel (të trajtuara në pjesën 2.8.3) e implementojnë ndarjen e mekanizmit dhe teknikës përmes disa blloqeve elementare ndërtuese. Këto blloqe pothuajse janë të pavarura nga teknika, duke mundësuar mekanizma dhe teknika më të avancuara të shtohen përmes moduleve të kernelit të krijuara nga shfrytëzuesi ose nga vetë programet e shfrytëzuesit. Për dallim, le të marrim Windows, një sistem operativ komercial shumë i popullarizuar në përdorim për më shumë se 30 vjet. Microsoft ka lidhur shumë mekanizmin dhe teknikën në sistem për të paraqitur një pamje të njëjtë në të gjitha pajisjet që përdorin sistemin operativ Windows. Të gjitha aplikacionet kanë interfejse të ngjashme, sepse vet interfejsi është ndërtuar në kernel dhe librari të sistemit. Apple ka adaptuar një strategji të njëjtë në sistemet e veta macOS dhe iOS.

Mund të bëjmë një krahasim të njëjtë në mes të sistemeve operative komerciale dhe me kod të hapur (open source). Për shembull, dallimi me Windows, i sistemit Linux, një sistem operativ me kod të hapur që ekzekutohet në disa pajisje kompjuterike dhe është në përdorim për më shumë se 25 vjet. Kerneli "standard" i Linux ka një algoritëm specifik për planifikimin e përdorimit të CPU-së, që është një mekanizëm që përkrahë një politikë të caktuar. Megjithatë, kushdo mund ndryshojë ose zëvendësojë planifikuesin (scheduler) për të përkrahur një politikë tjetër.

Vendimet e teknikave janë me rëndësi për dhënien në shfrytëzim të resurseve. Kurdo që është e nevojshme për të vendosur se a duhet të jepet në shfrytëzim një resurs ose jo, duhet të merret një vendim i teknikës. Kurdo që pyetja është *si* në vend të *çka*, duhet të përcaktohet një mekanizëm.

2.7.3 Implementimi

Sistemi operativ duhet patjetër të implementohet pasi të jetë dizajnuar. Meqë sistemet operative janë një grumbull i shumë programeve, të shkruara nga shumë njerëz gjatë një kohë të gjatë, është e vështirë të thuhet në përgjithësi se si ato implementohen.

Sistemet operative të hershme ishin shkruar një gjuhën asembler. Sot, shumica shkruhen në një gjuhë të lartë programimi si C ose C++, me pjesë të vogla të sistemit të shkruara në gjuhën asembler. Në të vërtetë, më shumë se një gjuhë e lartë programuese përdoret. Nivelet më të ulëta të kernelit mund të jenë të shkruara në gjuhën asembler dhe në C. Procedurat (funksionet) mund të jenë të shkruara në C dhe në C++, dhe libraritë e sistemit mund të jetë të shkruara në C++ ose madje edhe në gjuhë të lartë programuese. Android paraqet një shembull të mirë: kerneli i tij shkruhet pjesën më të madhe në C me disa pjesë në gjuhën assembly. Shumica e librarive të sistemit në Android shkruhen në C ose C++, përmes kornizave të tyre për aplikacione – që ofrojnë për programuesin një interfejs për sistem –

shkruhen kryesisht në Java. Arkitekturën e Android do të trajtojmë në më shumë detaje në pjesën 2.8.5.2.

Përparësitë e përdorimit të një gjuhe programuese të lartë, ose së paku një gjuhe për implementim të sistemeve, për implementim të sistemeve janë të njëjta me ato që fitohen kur gjuha përdoret për zhvillimin e programeve aplikative: kodi mund të shkruhet më shpejtë, është më kompakt dhe është më i lehtë për t'u kuptuar dhe për debugging. Përveç kësaj, përmirësimet në teknologjinë e kompajlerëve do të përmirësojnë kodin e krijuar për tërë sistemin operativ me një rikompajlim të thjeshtë. Në fund, një sistem operativ i shkruar në një gjuhë programuese të lartë bartet më lehtë në platforma tjera harduerike. Kjo është e një rëndësie të veçantë sidomos për sistemet operative që shkruhen për t'u ekzekutuar në disa sisteme harduerike të ndryshme, si në pajisjet e vogla embedded, sistemet Intel x86 dhe çipat ARM në telefonë dhe tabletë.

Mangësia e vetme e mundshme e implementimit të një sistemi operativ në një gjuhë programuese të lartë janë shpejtësia e zvogëluar dhe rritja e kërkesave për hapësirë të ruajtjes. Kjo, megjithatë, nuk është çështje në sistemet që përdorim sot. Edhepse një programues me përvojë në gjuhën asembler mund të shkruajë procedura të vogla efiçiente, për sistemet e mëdha një kompajler modern mund të kryejë analiza komplekse dhe aplikojë optimizime të sofistikuara që prodhojnë kod të mirë. Procesorët modern kanë pipeline të thellë (me shumë faza) dhe shumë njësi funksionale që mund të merren me detajet e varësive komplekse shumë më lehtë se sa mundet mendja e njeriut.

Siç ndodhë edhe në sisteme tjera, pjesa më e madhe e përmirësimeve në performancë të sistemeve operative mund të arrihen më shumë si rezultat i strukturave të të dhënave dhe algoritmeve më të mira se sa nga kodi i mirë në gjuhën asembler. Për më tepër, edhepse sistemet operative janë të mëdha, vetëm një pjesë e vogël e kodit është kritike për performancë të lartë; interrupt handlers (kodi që ekzekutohet për procesimin e interruptit), menaxhuesi për H/D, menaxhuesi i memories dhe planifikuesi (scheduler) i CPU-së janë procedurat më kritike. Pasi sistemi të shkruhet dhe është duke funksionuar në mënyrën e duhur, mund të identifikohen pjesët e dobëta (bottlenecks) dhe mund të rishkruhen për të punuar në mënyrë efikase.

2.8 Struktura e sistemit operativ

Një sistem i madh dhe kompleks si një sistem operativ modern duhet patjetër të ndërtohet me kujdes nëse duam që sistemi të funksionojë në mënyrën e duhur dhe të mund të ndryshohet lehtë. Një qasje e zakonshme është ndarja e punëve në pjesë të vogla ose module, në vend se të ndërtohet një sistem i vetëm. Secili prej këtyre moduleve duhet të jetë një pjesë e përcaktuar mirë e sistemit, me interfejse dhe funksione të përcaktuara mirë. Qasja e ngjashme mund të përdoret kur strukturojmë programe: në vend se të vendosim tërë kodin në funksionin main ( ), logjikën e ndajmë në disa funksione, duke qartësuar parametrat dhe vlerat e kthimit e pastaj të thërrasim ato funksione prej funksionit main ( ).

2.8.1 Struktura monolitike

Struktura më e thjeshtë për organizimin e një sistemi operativ është që të mos ketë fare strukturë. Kjo do të bëhej ashtu që i tërë funksionaliteti i kernelit do të vendosej në një fajll binar që ekzekutohet në një hapësirë të vetme të adresave. Kjo qasje – e njohur si struktura **monolitike** – është e zakonshme si teknikë për dizajnimin e sistemeve operative.

Një shembull i një strukturimi të kufizuar është versioni i parë i sistemit operativ UNIX, që përbëhet prej dy pjesëve të ndara: kerneli dhe programet e sistemit. Kerneli ndahet tutje në disa pjesë të interfejseve dhe drajverëve të pajisjeve, që janë shtuar e zgjeruar përgjatë viteve të zhvillimit të UNIX. Sistemin operativ tradicional UNIX mund të trajtojmë deri diku si të shtresuar, siç është paraqitur në Figura 2.12.

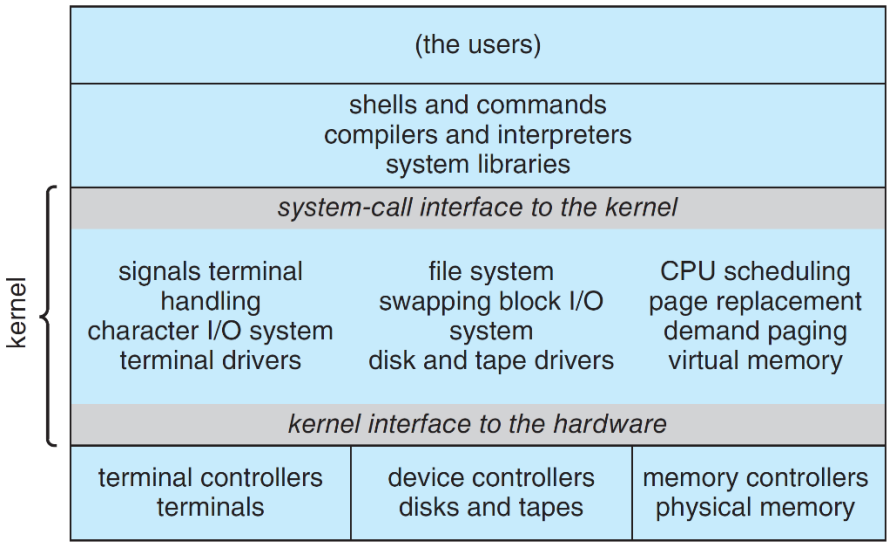


Figura 2.12 Struktura tradicionale e një sistemi UNIX

Çdo gjë përfundi interfejsit të thirrjeve sistemore dhe përmbi harduerin është kernel. Kerneli ofron sistemin e fajllave, planifikimin e CPU-së, menaxhimin e memories dhe funksione tjera të sistemit operativ përmes thirrjeve sistemore. Duke i marrë të gjitha së bashku, është një sasi shumë e madhe e funksionalitetit që duhet të kombinohet në një hapësirë të vetme të adresave.

Sistemi operativ Linux është bazuar në UNIX dhe i strukturuar ngjashëm, siç është paraqitur në Figura 2.13. Aplikacionet zakonisht përdorin librarinë standarde glibc të gjuhës C kur komunikojnë me kernel përmes interfejsit të thirrjeve sistemore. Kerneli i Linux është monolitik në kuptimin që ai ekzekutohet plotësisht në mënyrën e kernelit në një hapësirë të vetme të adresave, por siç do të trajtojmë në pjesën 2.8.4, ka një dizajn me module që lejon që kerneli të mund të ekzekutohet gjatë ekzekutimit.

Përkundër dukjes së thjeshtë të kernelëve monolitik, ata janë të vështirë për t'u implementuar dhe zgjeruar. Megjithatë, kernelët monolitik kanë një përparësi dalluese të performancës: ka pak punë që duhen kryer në interfejsin e thirrjeve sistemore dhe

komunikimi brenda në kernel është i shpejtë. Pra, përkundër mangësive të kernelëve monolitik, shpejtësia dhe efikasiteti i tyre bën që kjo strukturë të përdoret në UNIX, Linux dhe Windows.

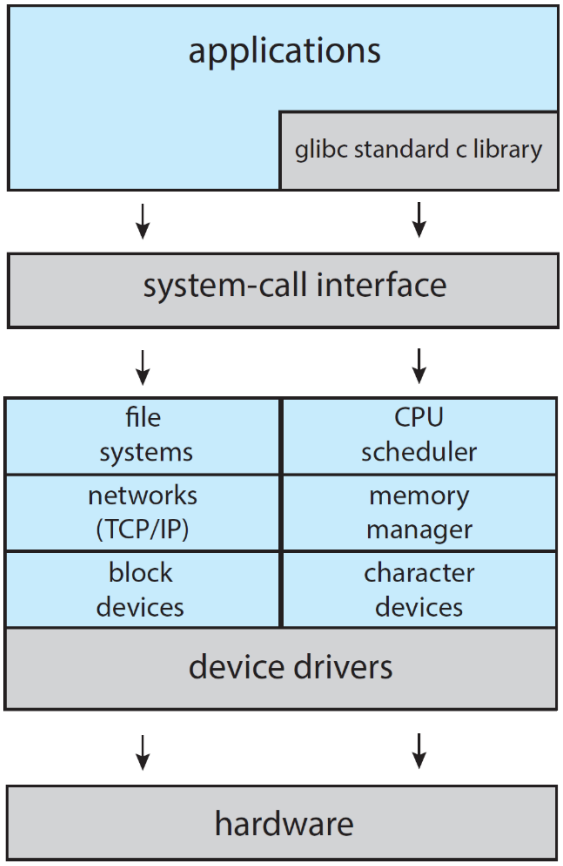


Figura 2.13 Struktura e sistemit Linux

2.8.2 Qasja me shtresa

Qasja monolitike shpesh njihet si një sistem **i lidhur fortë (tightly coupled)** sepse ndryshimet në njërën pjesë të sistemit mund të kenë ndikime të mëdha në pjesët tjera të sistemit. Për dallim, do të mund të dizajnonim një sistem **jo të lidhur fortë**. Një sistem i tillë ndahet në pjesë të vogla të ndara që kanë funksionalitet të caktuar e të kufizuar. Të gjitha këto pjesë së bashku formojnë kernelin. Përparësia e kësaj qasjeje me module është që ndryshimet në një pjesë kanë ndikim vetëm në atë pjesë dhe jo në pjesët tjera, duke u lejuar implementuesve të sistemit më shumë liri në krijimin dhe ndryshimin e mënyrës se si sistemi funksionon nga brenda.

Një sistem mund të bëhet modular në disa mënyra. Njëra metodë është **qasja me shtresa**, në të cilën sistemi ndahet në disa shtresa (nivele). Shtresa e poshtme (shtresa 0) është

hardueri; shtresa më e lartë (shtresa N) është interfejsi i shfrytëzuesit. Kjo strukturë me shtresa është paraqitur në Figura 2.14.

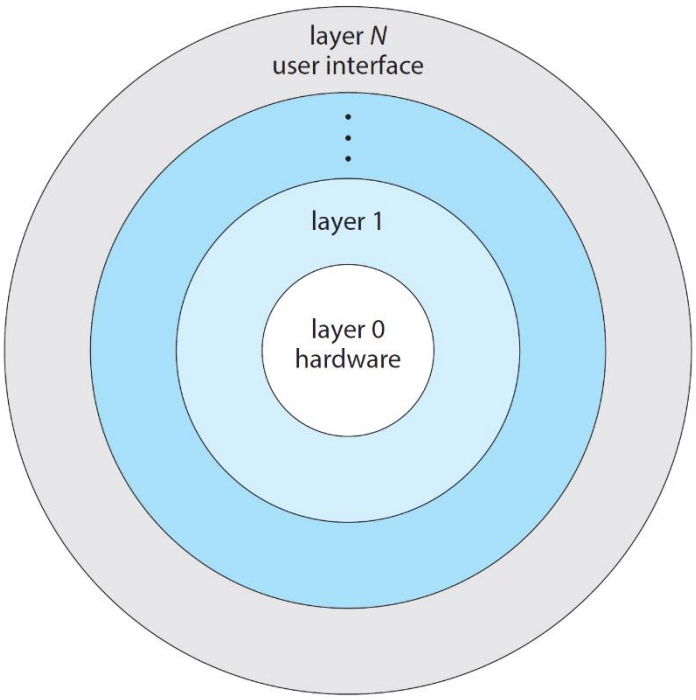


Figura 2.14 Një sistem operativ me shtresa

Një shtresë e sistemit operativ është një implementim i një objekti abstrakt të përbërë nga të dhëna dhe operacione që mund të punojnë me ato të dhëna. Një shtresë e sistemit operativ – të themi shtresa M – përbëhet nga struktura të të dhënave dhe një bashkësi e funksioneve që mund të thirren nga shtresat më të larta. Shtresa M, pastaj, mund të thërrasë operacionet në shtresat më të ulëta.

Përparësia kryesore e qasjes me shtresa është thjeshtësia në ndërtim dhe debugging. Shtresat përzgjedhen ashtu që secila përdorë funksione (operacione) dhe shërbime të vetëm shtresave më të ulëta. Kjo qasje thjeshton debugging dhe verifikimin e sistemit. Shtresa e parë mund të bëhet debugged pa u marrë me pjesën tjetër të sistemit, sepse, për nga definicioni, ajo përdorë vetëm harduerin bazë (që supozohet se është pa probleme) për të implementuar funksionet e saj. Kur shtresa e parë të jetë bërë debugged, mund të supozohet se ajo funksionon mirë përderisa shtresa e dytë të bëhet debugged e ashtu me radhë. Nëse gjendet ndonjë gabim gjatë debugging të një shtrese të caktuar, gabimi duhet patjetër të jetë në atë shtresë, për arsye se shtresa përfundi saj veçse është bërë debugged. Kështu, dizajnimi dhe implementimi i sistemit thjeshtohen.

Çdo shtresë implementohet vetëm me operacionet që ofrohen nga shtresat më të ulëta. Një shtresë nuk ka nevojë të di se si janë implementuar këto operacione; ka nevojë të di

vetëm se çfarë bëjnë ato operacione. Kështu, secila shtresë fshehë disa struktura të dhënash, operacione dhe harduer prej shtresave më të larta.

Sistemet me shtresa janë përdorë me sukses në rrjete kompjuterike (si TCP/IP) dhe në ueb aplikacione. Sido që të jetë, vetëm pak sisteme operative përdorin një qasje të pastër me shtresa. Një arsye janë sfidat për të përcaktuar në mënyrën e duhur funksionin e secilës shtresë. Përveç kësaj, performanca e përgjithshme e këtyre sistemeve është e dobët për arsye të punëve shtesë që nevojiten për të kërkuar nga programi i shfrytëzuesit që për të marrë një shërbim nga sistemi operativ duhet të kalojnë nëpër shumë shtresa. Megjithatë, sistemet operative moderne kanë disa shtresa. Në përgjithësi, këto sisteme kanë më pak shtresa me më shumë funksione, duke ofruar shumicën e përparësive të kodit të ndarë në module përderisa i shmangë problemet e përcaktimit të shtresës dhe ndërveprimin.

2.8.3 Mikrokernelët

Versioni i parë i sistemit operativ UNIX kishte një strukturë monolitike. Me zgjerimin e UNIX, kerneli u bë më i madh dhe më i vështirë për t'u menaxhuar. Në mesin e viteve 1980-ta, kërkuesit në Carnegie Mellon University zhvilluan një sistem operativ të quajtur **Mach** që e kishte kernelin të ndarë në module përmes qasjes me **mikrokernel**. Kjo metodë e strukturon sistemin operativ duke larguar të gjitha pjesët jo thelbësore prej kernelit në hapësira të ndryshme të adresave. Rezultati është një kernel më i vogël. Ka një pajtim të vogël për atë se cilat shërbime duhet të vendosen në kernel e cilat duhet të implementohen në hapësirën e shfrytëzuesit. Megjithatë, zakonisht, mikrokernelët ofrojnë menaxhim minimal të proceseve dhe të memories, krahas një mundësie për komunikim. Figura 2.15 paraqet arkitekturën e një mikrokerneli të zakonshëm.

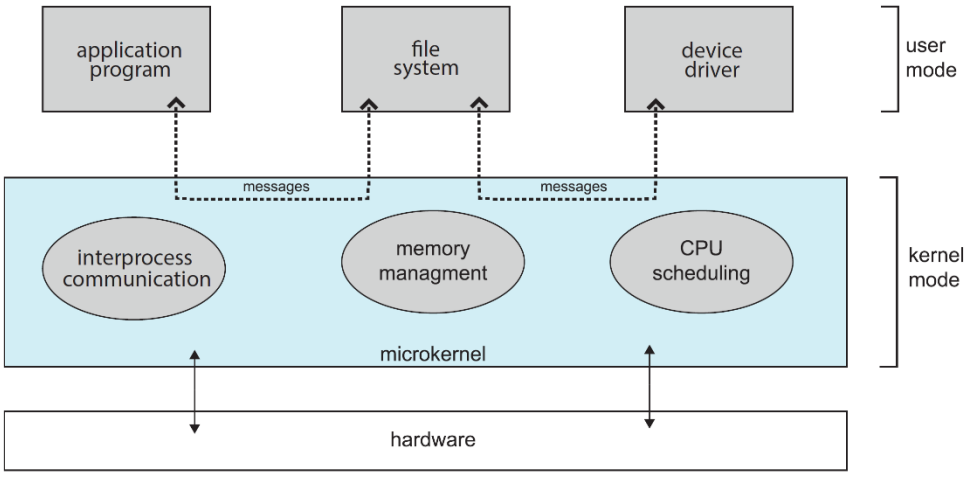


Figura 2.15 Arkitektura e një mikrokerneli të zakonshëm

Funksioni kryesor i mikrokernelit është të ofrojë komunikim në mes të programit të klientit dhe disa shërbimeve të cilat poashtu janë duke u ekzekutuar në hapësirën e shfrytëzuesit. Komunikimi bëhet përmes bartjes së mesazheve, kjo u trajtua te pjesa 2.3.3.5. Për shembull, nëse programi i klientit dëshiron të qaset në një fajll, duhet patjetër të komunikojë me

serverin e fajllave. Programi i klientit dhe shërbimi asnjëherë nuk komunikojnë direkt. Në vend të komunikimit direkt, ato komunikojnë në mënyrë indirekte duke shkëmbyer mesazhe me mikrokernelin.

Një përparësi e mikrokernelit është se e bën më të lehtë zgjerimin e sistemit operativ. Të gjitha shërbimet e reja shtohen në hapësirën e shfrytëzuesit dhe nuk kërkojnë ndryshime në kernel. Kur duhet të ndryshohet kernel, ndryshimet janë të vogla sepse mikrokerneli është një kernel i vogël. Sistemi operativ që fitohet është i lehtë për t'u bartur prej një dizajni të harduerit në një tjetër. Mikrokerneli ofron edhe më shumë siguri dhe besueshmëri, meqë shumica e shërbimeve ekzekutohen në hapësirën e shfrytëzuesit - në vend si procese në kernel. Nëse një shërbim dështon, pjesa tjetër e sistemit operativ mbetet e paprekur.

Mbase shembulli më i mirë i një sistemi operativ me mikrokernel është ***Darwin***, që është kerneli i sistemeve operative macOS dhe iOS. Në të vërtetë, Darwin, përbëhet prej dy kernelëve, njëri nga ta është mikrokerneli Mach.

Një shembull tjetër është QNX, një sistem operativ i kohës reale. Mikrokerneli Neutrino i QNX ofron shërbime për bartje të mesazheve dhe planifikim të proceseve. Merret edhe me detajet e vogla të komunikimit dhe me ndërprerjet (interrupts) harduerike. Të gjitha shërbimet tjera në QNX ofrohen përmes proceseve standarde që ekzekutohen jashtë kernelit në mënyrën e shfrytëzuesit.

Fatkeqësisht, performanca e mikrokernelëve mund të vuajë nga rritja e punëve në funksionet e sistemit. Kur dy shërbime të nivelit të shfrytëzuesit duhet patjetër të komunikojnë, mesazhet duhet patjetër të kopjohen në mes të shërbimeve, që ndodhen në hapësira të ndara të adresave. Për më tepër, sistemi operativ mund të ketë nevojë që të kalojë nga njëri proces në tjetrin për të shkëmbyer mesazhet. Puna që duhet kryer për të kopjuar mesazhet dhe kaluar në mes të proceseve ka qenë vështirësia më e madhe në rritjen e sistemeve operative të bazuara në mikrokernel. Le të marrim historinë e Windows NT: Versioni i parë kishte mikrokernel me organizim shtresor. Performanca e këtij versioni ishte më e vogël se e Windows 95. Windows NT 4.0 e kishte zgjidhur pjesërisht problemin e performancës duke bartur shtresat nga hapësira e shfrytëzuesit në hapësirën e kernelit dhe duke integruar ato më shumë. Kur ishte dizajnuar Windows XP, arkitektura e Windows-it ishte bërë më shumë monolitike se sa mikrokernel.

2.8.4 Modulet

Mbase metodologjia e tanishme më e mirë për dizajnim të sistemeve operative përfshinë përdorimin e **moduleve të kernelit të vendosshme në memorie (loadable kernel modules)**. Te ky dizajn, kerneli ka një bashkësi të pjesëve themelore dhe mund të lidhë (përdorë) shërbime shtesë përmes moduleve, gjatë kohës së startimit të sistemit ose gjatë kohës së ekzekutimit. Ky lloj i dizajnit është i zakonshëm në implementimet moderne të UNIX, si Linux, macOS dhe Solaris, poashtu edhe Windows.

Ideja e dizajnit është që kerneli të ofrojë shërbime themelore, përderisa shërbimet tjera implementohen në mënyrë dinamike, në kohën kur kerneli është duke u ekzekutuar. Lidhja e shërbimeve në mënyrë dinamike është më e preferueshme se shtimi i karakteristikave të reja

direkt në kernel, gjë që do të kërkonte rikompajlimin e kernelit sa herë që do të bëhej ndonjë ndryshim. Kështu, për shembull, do të mund të ndërtonim algoritmet për planifikimin e CPU-së dhe menaxhimin e memories direkt në kernel dhe pastaj të shtonim përkrahje për sisteme të ndryshme të fajllave me anë të moduleve të vendosshme në memorie.

Rezultati përfundimtar do të ishte një sistem me shtresa në të cilin secila pjesë e kernelit ka interfejse të përcaktuara dhe të mbrojtura mirë; por është më fleksibil se një sistem me shtresa, sepse secili modul mund të thërrasë cilindo modul tjetër. Qasja është e ngjashme me qasjen përmes mikrokernelit në atë se moduli kryesor ka vetëm disa funksione themelore dhe njohuritë se si t'i vendosë në memorie dhe të komunikojë me modulet tjera; por është më efiçiente, sepse modulet nuk kanë nevojë të përdorin shkëmbimin e mesazheve për të komunikuar.

Linux përdorë module të kernelit të vendosshme në memorie, kryesisht për të përkrahur drajverët e pajisjeve dhe sistemet e fajllave. Këto module mund të "futen" në kernel gjatë startimit të sistemit ose gjatë kohës së ekzekutimit, sikur në rastin kur të vendoset një pajisje USB në një kompjuter që është duke u ekzekutuar. Nëse kerneli i Linux-it nuk e ka drajverin e nevojshëm, ai drajver mund të vendoset në memorie në mënyrë dinamike. Për Linux-in, modulet mundësojnë kernel dinamik dhe modular, duke ruajtur përparësitë (përfitimet) e performancës të një sistemi monolitik.

2.8.5 Sistemet hibride

Në praktikë, pak sisteme operative kanë një strukturë të vetme të përcaktuar mirë. Për dallim, ato kombinojnë struktura të ndryshme, duke ndërtuar sisteme që merren me çështjet e performancës, sigurisë dhe përdorshmërisë. Për shembull, Linux është monolitik, sepse vendosja e sistemit operativ në një hapësirë të adresave ofron performancë shumë efiçiente. Megjithatë, është edhe modular, ashtu që funksione të reja mund të shtohen në kernel në mënyrë dinamike. Windows është poashtu shumë monolitik (përsëri për arsye të performancës), por ka disa karakteristika tipike të sistemeve me mikrokernel, përfshirë ofrimin e përkrahjes të nënsistemeve të ndara (të njohura si *personalitete* të sistemit operativ) që ekzekutohen si procese në hapësirën e shfrytëzuesit. Sistemet Windows ofrojnë edhe përkrahje për modulet e kernelit të vendosshme në memorie.

2.8.5.1 macOS dhe iOS

Sistemi operativ macOS i Apple është dizajnuar që të ekzekutohet kryesisht në sistemet kompjuterike desktop dhe laptop, ndërsa iOS është një sistem operativ mobil i dizajnuar për telefonin e mençur iPhone dhe kompjuterin tablet iPad. Për nga arkitektura, macOS dhe iOS kanë shumë gjëra të përbashkëta dhe do t'i trajtojmë bashkë, duke theksuar se çfarë kanë të përbashkët si dhe dallimet në mes tyre. Arkitektura e përgjithshme e këtyre dy sistemeve është paraqitur në Figura 2.16. Karakteristikat e disa shtresave përfshijnë:

- **Shtresa e përvojës së shfrytëzuesit**. Kjo shtresë përcakton interfejsin softuerik që lejon shfrytëzuesit të përdorin pajisjet kompjuterike. macOS përdorë interfejsin ***Aqua***, që është dizajnuar për muas ose trackpad (në laptop), ndërsa iOS përdorë interfejsin ***Springboard,*** që është dizajnuar për pajisje me ekran me prekje.

- **Shtresa e kornizave (frameworks) të aplikacionit**. Kjo shtresë përfshinë kornizat ***Cocoa*** dhe ***Cocoa Touch***, që ofrojnë një API për gjuhët programuese Objective-C dhe Swift. Dallimi kryesor në mes Cocoa dhe Cocoa Touch është se e para përdoret për zhvillimin e aplikacioneve për macOS dhe e dyta në iOS për të ofruar përkrahje për karakteristika harduerike për pajisje mobile, sikur ekrane me prekje.
- **Kornizat themelore**. Kjo shtresë përcakton kornizat që përkrahin grafikë dhe media përfshirë, QuickTime dhe OpenGL.
- **Ambienti (rrethina) i kernelit**. Ky ambient, i njohur si **Darwin**, përfshinë mikrokernelin Mach dhe kernelin BSD UNIX.

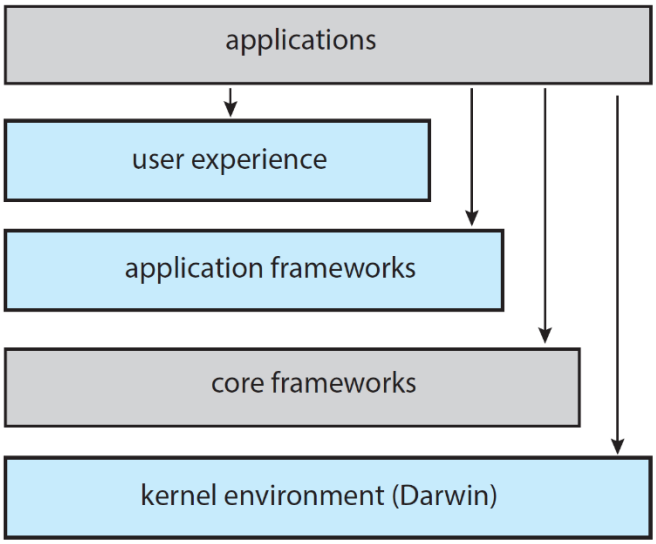


Figura 2.16 Arkitektura e sistemeve operative macOS dhe iOS të Apple

Siç është paraqitur në Figura 2.16, aplikacionet mund të dizajnohen me përparësi për karakteristikat e përvojës së shfrytëzuesit ose të shmangin ato dhe komunikojnë direkt me kornizën e aplikacionit ose me kornizën themelore. Për më tepër, një aplikacion mund të anashkalojë kornizat plotësisht dhe komunikojë direkt me ambientin e kernelit. (Një shembull i këtij rasti është një program i shkruar pa ndonjë interfejs të shfrytëzuesit në gjuhën C i cili thërret thirrjet sistemore POSIX).

Disa dallime të theksuara në mes macOS dhe iOS përfshijnë këto:

- Meqë macOS është ndërtuar për sisteme kompjuterike desktop dhe laptop, kompajlohet që të ekzekutohet në arkitektura Intel. iOS është dizajnuar për pajisje mobile dhe kompajlohet për arkitekturat e bazuara në ARM. Ngjashëm, kerneli i iOS është ndryshuar për të ofruar disa karakteristika dhe nevoja specifike të sistemeve mobile, si menaxhimi i rrymës dhe menaxhimi i memories. Përveç kësaj, iOS ka më shumë karakteristika të sigurisë se macOS.
- Sistemi operativ iOS në përgjithësi është më i kufizuar për zhvilluesit se macOS dhe madje mund të jetë i mbyllur për zhvilluesit. Për shembull, iOS kufizon qasjen në POSIX dhe API-të BSD në iOS, përderisa janë në dispozicion të zhvilluesve në macOS.

Në vazhdim do të trajtojmë më gjerësisht Darwin, që përdorë një strukturë hibride. Darwin është një sistem shtresor që përbëhet kryesisht nga mikrokerneli Mach dhe kerneli UNIX BSD. Struktura e Darwin është paraqitur në Figura 2.17.

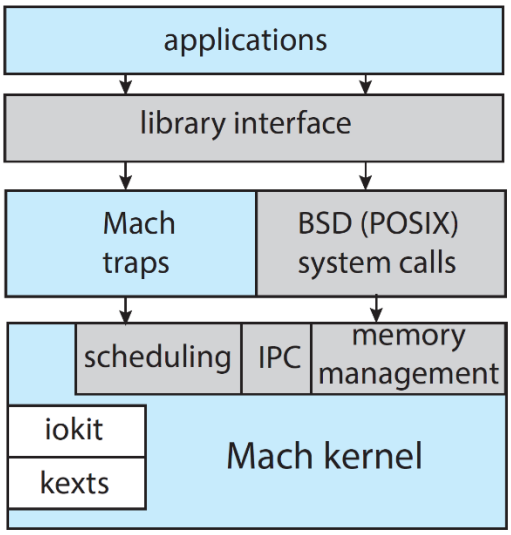


Figura 2.17 Struktura e Darwin

Përderisa shumica e sistemeve operative ofrojnë një interfejs të vetëm për kernelin me anë të thirrjeve sistemore – përmes librarisë standarde të gjuhës C në sistemet UNIX dhe Linux – Darwin ofron dy interfejse për thirrjet sistemore: thirrjet sistemore për Mach (të njohura si **traps**) dhe thirrjet sistemore BSD (që ofrojnë funksionalitet POSIX). Interfejsi i këtyre thirrjeve sistemore është një bashkësi e pasur e librarive që përfshijnë jo vetëm librarinë standarde të gjuhës C, por edhe librari për komunikime në rrjete, siguri dhe përkrahje për gjuhët programuese.

Përfundi interfejsit të thirrjeve sistemore, Mach ofron shërbimet më themelore të sistemit operativ, përfshirë menaxhimin e memories, planifikimin e CPU-së dhe mundësitë për komunikim në mes të proceseve (inter-process communication – IPC) si bartja e mesazheve dhe thirrja e procedurave në distancë (remote procedure call). Shumë nga funksionet e ofruara nga Mach janë në dispozicion përmes **abstrahimeve të kernelit**, që përfshijnë punë (një proces i Mach), threda, objekte të memories dhe porte (që përdoren për IPC). Si shembull, një aplikacion mund të krijojë një proces të ri përmes thirrjes sistemore fork ( ) në BSD POSIX. Mach përdorë një abstrahim në kernel për punën për paraqitjen e procesit në kernel.

Përveç Mach dhe BSD, ambienti i kernelit ofron një pako të H/D për zhvillimin e drajverëve të pajisjeve dhe moduleve që mund të vendosen në mënyrë dinamike në memorie (të cilave macOS u referohet si **kernel extensions** ose **kexts**).

Në pjesën 2.8.3 trajtuam se si puna që duhet të kryhet për bartje të mesazheve në mes të shërbimeve të ndryshme që ekzekutohen në hapësirën e shfrytëzuesit ndikon negativisht në performancën e mikrokernelit. Për t'u marrë me këto probleme të performancës, Darwin kombinon Mach, BSD, dhe pakon për H/D dhe pjesët tjera të kernelit në një hapësirë të vetme të

adresave. Kështu, Mach nuk është një mikrokernel i pastër në kuptimin që disa nënsisteme ekzekutohen në hapësirën e shfrytëzuesit.

Apple e ka publikuar sistemin operativ Darwin me kod të hapur. Si rezultat i kësaj, shumë projekte kanë shtuar funksione shtesë në Darwin, sikur sistemi për dritare X-11 dhe përkrahja për disa sisteme shtesë të fajllave. Për dallim prej Darwin, interfejsi Cocoa, sikur edhe disa korniza të mbyllura të Apple për zhvillimin e aplikacioneve për macOS janë me kod të mbyllur.

2.8.5.2 Android

Sistemi operativ ishte dizajnuar nga Open Handset Alliance (udhëhequr kryesisht prej Google) dhe ishte zhvilluar për telefonët e mençur dhe kompjuterët tabletë Android. Përderisa iOS është dizjanuar për t'u ekzekutuar në pajisjet mobile Apple dhe është me kod të mbyllur, Android ekzekutohet në shumë platforma të ndryshme mobile dhe është me kod të hapur, kjo pjesërisht tregon rritjen e shpejtë të popullaritetit të Android. Struktura e Android është paraqitur në Figura 2.18.

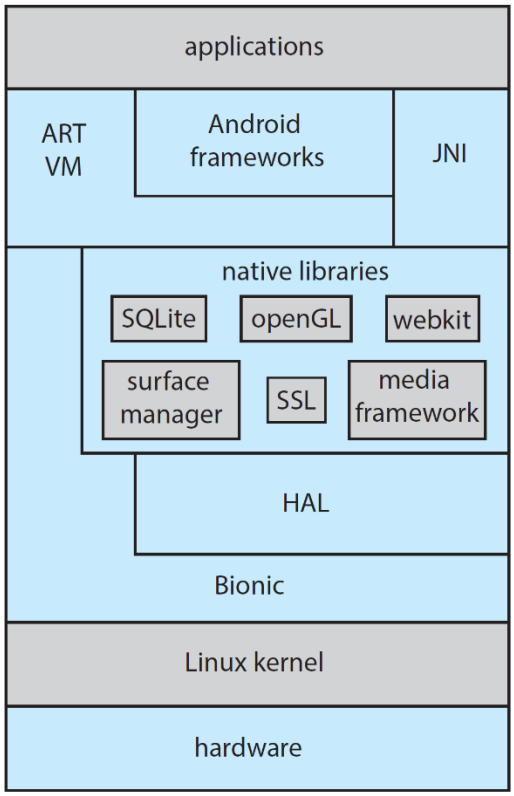


Figura 2.18 – Struktura e Android të Google

Android është i ngjashëm me iOS në atë se është një strukturë me disa shtresa të softuerit të cilat ofrojnë një bashkësi të pasur të kornizave që përkrahin grafikë, audio dhe karakteristika harduerike. Këto karakteristika, në anën tjetër, ofrojnë një platformë për zhvillimin e aplikacioneve mobile që ekzekutohen në një numër të madh të pajisjeve që përdorin sistemin Android.

Dizajnuesit e softuerit për pajisje Android zhvillojnë aplikacionet përmes gjuhës programuese Java, por dizajnuesit në përgjithësi nuk përdorin API standard të Java. Google ka dizajnuar një API të veçantë për Android për zhvillim përmes Java. Aplikacionet e zhvilluara në Java kompajlohen në një formë që mund të ekzekutohet në rrethinën për ekzekutim Android RunTime ART, një makinë virtuale e dizajnuar për Android dhe e optimizuar për pajisje mobile me memorie dhe mundësi të përpunimit përmes CPU të kufizuar. Programet në Java së pari kompajlohen në bytecode në fajlla .class dhe pastaj përkthehen në një fajll të ekzekutueshëm .dex. Përderisa shumë makina virtuale të Java-s bëjnë kompajlimin e kodit vetëm në kohën kur të thirret për ekzekutim (**just-in-time - JIT**) për të përmirësuar efiçiencën e aplikacionit, ART e bën kompajlimin para kohe (**ahead-of-time – AOT)**. Fajllat .dex kompajlohen në nivelin e gjuhës së makinës kur të instalohen në një pajisje, prej nga mund të ekzekutohen në ART. AOT mundëson ekzekutim më efikas të aplikacionit e poashtu edhe zvogëlim të konsumit të rrymës, karakteristika këto që janë qenësore për sistemet mobile.

Zhvilluesit për Android mund të shkruajnë edhe programe në Java që përdorin interfejsin e Java-s – që lejon zhvilluesit të anashkalojnë makinën virtuale dhe të shkruajnë programe në Java që mund të qasen në karakteristika (mundësi) të caktuara harduerike. Programet e shkruara duke përdorë interfejsin e Java-s në përgjithësi nuk mund të barten nga një pajisje harduerike në një tjetër.

Bashkësia e librarive për aplikacione Android përfshinë kornizat për zhvillimin e ueb shfletuesve (webkit), përkrahje për baza të të dhënave (SQLite) dhe përkrahje për rrjete, si secure sockets (SSL).

Meqë Android mund të ekzekutohet në pothuajse në një numër të pakufizuar të pajisjeve harduerike, Google ka vendosur që të abstrahojë (fshehë) harduerin fizik përmes një shtrese për abstrahim të harduerit (hardware abstraction layer - HAL). Duke abstrahuar të gjithë harduerin, si kamerën, çipin e GPS dhe sensorë tjerë, HAL u ofron aplikacioneve një pamje të njëjtë të pavarur nga hardueri specifik.

Libraria standarde e gjuhës C që përdoret në sistemet Linux është libraria GNU C (glibc). Për Android, Google zhvilloi standardin **Bionic** të librarisë për gjuhën C. Jo vetëm që Bionic harxhon më pak memorie se glibc, por është dizajnuar edhe për CPU-të e ngadalshëm që janë në pajisje mobile. (Për më tepër, Bionic lejon Google të anashkalojë licencën GPL të glibc).

Në fund të strukturës shtresore të softuerit të Android është një kernel Linux. Google ka ndryshuar kernelin Linux që përdoret në Android në shumë aplikime për të përkrahur nevojat e veçanta të sistemeve mobile, sikur menaxhimi i rrymës. Kishin bërë ndryshime edhe në menaxhim të memories dhe ndarje të memories, ka shtuar edhe një formë të re të komunikimit në mes të proceseve (IPC) të njohur si ***Binder***.

2.9 Ndërtimi dhe startimi i një sistemi operativ

Mund të dizajnohet, kodohet dhe implementohet një sistem operativ në mënyrë specifike për një konfigurim të caktuar të një kompjuteri. Është më e zakonshme që sistemet operative të dizajnohen që të ekzekutohen në një klasë të kompjuterëve që kanë konfigurime të ndryshme të pajisjeve periferike.

2.9.1 Ndërtimi i sistemit operativ

Një sistem kompjuterik, zakonisht kur të blehet e ka të instaluar një sistem operativ. Për shembull, ju mund të bleni një laptop të ri që ka të instaluar sistemin operativ Windows ose macOS. Le të themi se ju dëshironi të ndërroni sistemin operativ ose të shtoni një tjetër. Kompjuteri që blejmë mund të mos ketë të instaluar asnjë sistem operativ. Në rastin kur ka të instaluar një sistem operativ janë disa mundësi të vendosjes së sistemit operativ përkatës në kompjuter dhe konfigurimit të tij.

Për të ndërtuar një sistem operativ nga fillimi, duhet patjetër të ndjeken këta hapa:

1. Shkrimi i kodit të sistemit operativ (ose marrja e një kodi që kanë shkruar tjerët).
2. Konfigurimi i sistemit operativ për sistemin në të cilin do të ekzekutohet.
3. Kompajlimi i sistemit operativ.
4. Instalimi i sistemit operativ.
5. Startimi i kompjuterit dhe sistemit operativ të ri.

Konfigurimi i sistemit përfshinë përcaktimin se cilat karakteristika (mundësi) do të përfshihen, kjo varet prej sistemit operativ. Zakonisht, parametrat që përshkruajnë se si konfigurohet sistemi ruhen në një fajll të konfigurimit të një lloji të caktuar, dhe pasi të krijohet ky fajll, mund të përdoret në shumë mënyra.

Në një rast ekstrem, një administrues sistemi mund të përdorë atë fajll për të ndryshuar një kopje të kodit burimor të sistemit operativ. Pastaj bëhet kompajlimi i plotë i sistemit operativ (që njihet si **system build**). Deklarimi i të dhënave, inicializimi dhe konstantet, krahas kompajlimit, prodhojnë një version të sistemit operativ në formë binare që përshtatet për sistemin që përshkruhet në fajllin e konfigurimit.

Në një nivel më pak të përshtatur, përshkrimi i sistemit mund të dërgojë në përzgjedhjen nga një librari ekzistuese të moduleve binare të parakompajluara. Këto module lidhen sëbashku për të ndërtuar sistemin operativ. Kjo lejon që libraria të përmbajë drajverët e pajisjeve për të gjitha pajisjet për H/D që përkrahen, por vetëm ato që nevojiten përzgjedhen dhe lidhen në sistem operativ. Meqë sistemi nuk rikompajlohet, ndërtimi i sistemit është i shpejtë, por sistemi që fitohet mund të jetë paksa i përgjithshëm dhe mund të mos përkrahë konfigurime të ndryshme të harduerit.

Në rastin tjetër ekstrem, është e mundur të ndërtohet një sistem që është plotësisht modular. Këtu, përzgjedhja bëhet në kohën e ekzekutimit e jo në kohën e kompajlimit ose të lidhjes. Ndërtimi i sistemit kërkon thjeshtë vendosjen e vlerave të parametrave që përshkruajnë konfigurimin e sistemit.

Dallimet kryesore në mes të këtyre qasjeve janë madhësia dhe sa është i përgjithshëm sistemi i fituar dhe lehtësia e ndryshimit të tij kur të ndryshojnë konfigurimet e harduerit. Për sistemet embedded, nuk është e rallë që të përdoret qasja e parë dhe të krijohet një sistem operativ për një konfigurim hardueri që nuk ndryshon. Megjithatë, shumica e sistemeve operative moderne që përkrahin kompjuterët desktop, laptop dhe pajisjet mobile përdorin qasjen e dytë. Te kjo qasje, sistemi operativ ndërtohet për një konfigurim të caktuar të harduerit, por përdorimi i teknikave si modulet e kernelit të vendosshme në memorie ofron përkrahje për ndryshime dinamike të sistemit.

Në vazhdim do të tregojmë se si të ndërtohet prej fillimit një sistem Linux, aty është e nevojshme të kryhen këta hapa:

1. Shkarkimi i kodit burimor të Linux-it prej http://www/kernel.com
2. Konfigurimi i kernelit përmes komandës “make menuconfig”. Ky hap krijon fajllin .config të konfigurimit.
3. Kompajlimi i pjesës kryesore të kernelit përmes komandës “make”. Komanda “make” e kompajlon kernelin bazuar në parametrat e konfigurimit që merren prej fajllit .config., duke prodhuar fajllin vmlinuz, që paraqet kernelin.
4. Kompajlimi i moduleve të kernelit përmes komandës “make modules”. Sikur te kompajlimi i kernelit, kompajlimi i moduleve varet prej parametrave të përcaktuar në fajllin .config.
5. Instalimi i moduleve të kernelit në vmlinuz përmes komandës “make modules_install".
6. Instalimi i kernelit në sistem përmes komandës “make install”.

Kur të ristartohet sistemi, sistemi do të ekzekutojë sistemin e ri operativ.

Në një mënyrë tjetër, është e mundur të ndryshohet një sistem operativ ekzistues duke instaluar një makinë virtuale për Linux. Kjo do të lejojë që sistemi operativ nikoqir (si Windows ose macOS) të ekzekutojë Linux-in.

Janë disa mundësi për instalimin e Linux-it si makinë virtuale. Një mundësi është të ndërtohet një makinë virtuale prej fillimit. Kjo mundësi është e ngjashme me ndërtimin e një sistemi Linux prej fillimit; megjithatë, nuk është e nevojshme të kompajlohet sistemi operativ. Një qasje tjetër është përdorimi i një ndonjë vegle për makinën virtuale të Linux, që është një sistem operativ i ndërtuar dhe konfiguruar. Ky opcion kërkon që thjeshtë të shkarkohet dhe instalohet vegla përmes ndonjë softueri për virtualizim si VirtualBox ose VMware.

2.9.2 Startimi i sistemit

Pasi të ndërtohet një sistem operativ, duhet patjetër të jetë në dispozicion që të përdoret nga hardueri. Si e di hardueri se ku është kernel ose si të vendosë kernelin në memorie? Procesi i startimit të një kompjuteri duke vendosur kernelin në memorie njihet si **startimi (booting)** i sistemit. Në shumicën e sistemeve, startimi bëhet kështu:

1. Një program i vogël i njohur si **bootstrap** ose **boot loader** e gjenë kernelin.
2. Kerneli vendoset në DRAM dhe startohet.
3. Kerneli e starton harduerin.
4. Sistemi i fajllave bëhet pjesë e sistemit.

Disa sisteme kompjuterike përdorin një rrugë të startimit me shumë hapa: kur të lëshohet në punë kompjuteri, ekzekutohet një boot loader i vogël i njohur si **BIOS** i ruajtur në memorie të qëndrueshme. Ky boot loader fillestar, zakonisht nuk bën gjë tjetër përveç vendosjes në memorie të një boot loader tjetër, që ndodhet në një pjesë që nuk ndryshon të diskut të quajtur **boot block**. Programi që ndodhet në boot block mund të jetë i tillë që ka mundësinë të vendosë në memorie tërë sistemin operativ dhe të fillojë ekzekutimin e tij. Zakonisht, është kod i thjeshtë (meqë duhet

patjetër të vendoset vetëm në një bllok të diskut) dhe e di vetëm adresën në disk dhe gjatësinë e pjesës tjetër të programit bootstrap.

Shumica e sistemeve kompjuterike të fundit kanë zëvendësuar me UEFI (Unified Extensible Firmware Interface) procesin e startimit të bazuar në BIOS. UEFI ka disa përparësi ndaj BIOS-it, përfshirë përkrahjen më të mirë për sistemet 64 bitëshe dhe disqe të mëdha. Mbase përparësia kryesore është që UEFI është një menaxhues i vetëm i plotë i startimit dhe në atë mënyrë është më i shpejtë se procesi me shumë hapa përmes BIOS.

Pavarësisht startimit përmes BIOS ose UEFI, programi bootstrap mund të kryejë disa punë. Përveç vendosjes në memorie të fajllit që përmban kernelin, ai ekzekuton edhe testime për të përcaktuar gjendjen e kompjuterit – për shembull, testimi i memories, CPU-së dhe njohja e pajisjeve. Nëse kalohet testi, programi mund të vazhdojë me hapat e startimit. Programi bootstrap mund poashtu të startojë të gjitha aspektet e sistemit, që nga regjistrat e CPU-së e deri te kontrolluesit e pajisjeve dhe përmbajtjet e memories. Herët ose vonë, ai starton sistemin operativ dhe shton sistemin e fajllave. Vetëm pasi të kryhet kjo pjesë e punës mund të thuhet që sistemi është duke u **ekzekutuar**.

GRUB është një program bootstrap me kod të hapur për sistemet Linux dhe UNIX. Vlerat e parametrave të startimit për sistemin vendosen në një fajll konfigurimi të GRUB, i cili vendoset në memorie gjatë startimit. GRUB është fleksibil dhe lejon të bëhen ndryshime gjatë kohës së startimit, përfshirë ndryshimin e parametrave të kernelit dhe madje edhe zgjedhjen në mes të kernelëve të ndryshëm që mund të startohen. Si shembull, në vazhdim kemi disa parametra nga fajlli /proc/cmdline në Linux, që përdoret gjatë startimit:

```
BOOT_IMAGE = /boot/vmlinuz-4.4.0.-59-generic

root = UUID = 5f2e2232-4e47-4fe8-ae94-45ea749a5c92
```

`BOOT_IMAGE` është emri i kernelit që duhet të vendoset në memorie dhe `root` paraqet një identifikues unik të sistemit të fajllave.

Për të kursyer memorien si dhe për të zvogëluar kohën e startimit, kerneli i Linux-it është një fajll i kompresuar (duke zvogëluar madhësinë) i cili kthehet në madhësinë e tij pasi të vendoset në memorie. Gjatë procesit të startimit, zakonisht boot loader krijon një sistem RAM të përkohshëm të fajllave, i njohur si `initramfs`. Ky sistem i fajllave përmban drajverët dhe modulet e nevojshme të kernelit që duhet patjetër të instalohen për të përkrahur sistemin kryesor të fajllave (që nuk është në memorie). Pasi kerneli ka startuar dhe drajverët e nevojshëm janë instaluar, kerneli e ndërron sistemin kryesor të fajllave nga vendi i përkohshëm RAM në vendin përkatës të sistemit të fajllave. Në fund, Linux krijon procesin `system`, procesi i parë në sistem dhe pastaj starton shërbime tjera (për shembull, një ueb server dhe/ose një bazë të dhënash). Krejt në fund, sistemi do t'i ofrojë shfrytëzuesit mundësinë që të logohet në sistem.

Është me rëndësi të thuhet që mekanizmi i startimit nuk është i pavarur nga boot loader. Kështu që, janë versione specifike të boot loader GRUB për BIOS dhe UEFI dhe programi firmware duhet patjetër të di se cili boot loader do të përdoret.

Procesi i startimit për sistemet mobile është pak më ndryshe nga ai i kompjuterëve tradicional. Për shembull, edhepse kerneli i tij është i bazuar në Linux, Android nuk përdorë GRUB dhe në vend të tij ua len prodhuesve që të ofrojnë boot loaders. Boot loader më i njohur për Android është LK ("little kernel"). Sistemet Android përdorin të njëjtin kernel të kompresuar sikur Linux, poashtu edhe një sistem RAM fillestar të fajllave. Megjithatë, përderisa Linux heq dorë nga `initramfs` pasi që të gjithë drajverët e nevojshëm të jenë vendosur në memorie, Android vazhdon të përdorë `initramfs` si sistem kryesor për fajlla. Pasi kerneli të jetë vendosur në memorie dhe sistemi kryesor i fajllave të jetë shtuar në sistem, Android starton procesin `init` dhe krijon një numër të shërbimeve para se të paraqesë në ekran pamjen e desktopit (home screen).

Në fund, boot loaders për shumicën e sistemeve operative - përfshirë Windows, Linux dhe macOS, si dhe iOS dhe Android - ofrojnë mundësinë që të startohet në **recovery mode** ose **single-user mode** për të testuar në lidhje me probleme të harduerit, përmirësimi i gabimeve në sisteme të fajllave e madje edhe riinstalimin e sistemit operativ. Përveç dështimeve të harduerit, sistemet kompjuterike mund të përballen edhe me gabime në softuer dhe performancë të dobët të sistemit operativ.

Referenca

Materiali është marrë nga kapitulli 2 (faqe 55) i librit **Operating System Concepts** *i autorëve* **Abraham Silberschatz, Peter Baer Galvin dhe Greg Gagne.**