Curs 08: Fire de execuție

Sumar curs anterior

un sistem este sigur dacă funcţionează conform specificaţiilor
o componentă este securitatea aplicaţiilor la rulare (runtime application security)
o parte importantă este securitatea memoriei
spaţiul virtual de adrese al procesului este compus din zone de memorie cu permisiuni diferite: read-write, read-only, read-executable
o aplicaţie are un flux de execuţie descris de control flow graph (CFG)
există două tipuri de atacuri: atacuri ce adaugă noi noduri în graf (code injection attacks) sau care adaugă arce şi refolosesc graful în moduri benefice atacatorului (code reuse attacks)
vulnerabilităţile “standard” sunt buffer overflow şi out of bounds errors
un atacator urmăreşte suprascrierea de informaţii pentru a modifica CFG-ul programului
interesant este de realizat stack buffer overflow attacks, pentru că se suprascrie adresa de retur
adresa de retur este un code pointer, un pointer la o zonă de cod; suprascrierea unui code pointer oferă atacatorului posibilitatea controlului fluxului de execuţie (control flow hijack)
partea de code injection presupune injectarea unei secvenţe de cod în memoria procesului: shellcode
mecanisme de protecţie sunt: stack guard, data execution prevention, address space layout randomization, control flow integrity
în general se urmăreşte schema: vulnerabilitate/problemă, atac, metodă preventivă, bypass la metodă preventivă
+ demo de atac/bypass de Stack Smashing Protection

Tipuri de acțiuni în sisteme de calcul

sistemele de calcul oferă sprijin pentru realizarea de acţiuni ale utilizatorilor
o acțiune: obținerea unor date de intrare, prelucrarea lor, furnizarea unor date de ieșire (sau părţi din acestea)
exemple:
* un server web obţine date de pe disc (fişier, bază de date) şi la transferă la reţea (I/O intensive); un server web cu procesare (de exemplu modul PHP sau Ruby sau Python) are şi o fază de procesare (preponderent I/O intensive)
* un editor în general obţine date de la tastatură şi le scrie pe disc într-un fişier; face prelucrare atunci când face căutări sau înlocuiri (preponderent I/O intensive cu burst-uri de CPU intensive)
* utilitarul grep obţine date din fişier, caută prin ele expresii regulate şi scrie rezultatul în alt fişier (CPU intensive)
* un procesor video (parte dintr-un player sau editor video) face encoding/decoding la datele dintr-un fişier (CPU intensive)
acțiunile (sau sub-acțiunile) sunt I/O intensive sau CPU intensive

I/O intensive: operează des cu discul, placa de reţea sau alte forme de I/O
CPU intensive: folosesc des procesorul

Sisteme de calcul multi-core / multi-procesor

mai multe procesoare ne permit să:
* rulăm mai multe acţiuni diferite, câte una pe procesor
* să rulăm aceeaşi acţiune în paralel pe mai multe procesoare
în general o acţiune este abstractizată de un proces, care este planificat pe un procesor
pe sistemele de operare multi-procesor putem rula mai multe procese diferite, sporind astfel productivitatea sistemului (throughput); alternativa ar fi să rulăm pe rând fiecare proces (multitasking, scheduling): un procesor rulează un editor, altul rulează un server web, altul un procesor video, altul un browser
pentru a rula aceeaşi acţiune în paralel pe mai multe procesoare, avem opţiunea să creăm mai multe procese de acelaşi tip (prin fork): mai multe procese de tip server web, fiecare servind câte o conexiune; mai multe procese de tip procesor video, fiecare lucrând pe o parte din datele de intrare
dezavantaje folosire procese
* pentru aceeaşi acţiune ai nevoie de date comune, mai dificil de partajat (se poate cu memorie partajată, dar e nevoie de un efort)
* overhead de memorie pentru fiecare proces creat: tabelă de pagini nouă, structuri interne în sistemul de operare
* overhead-ul de creare nu e atât de mare: se creează o tabelă de pagini nouă şi structuri interne (se face relativ repede)
* overhead de schimbare de context: schimbare de tabele de pagini, flush la TLB

Thread-uri (fire de execuţie)

thread-urile sunt o variantă simplă a proceselor pentru executarea de acţiuni: se mai numesc lightweight processes (LWP)
un proces are unul sau mai multe thread-uri
thread-urile sunt o abstractizare doar pentru procesor (acţiune); spaţiul virtual de adrese, descriptori de I/O aparţin procesului şi sunt informaţii partajate între thread-uri
thread-urile compensează dezavantajele folosirii proceselor:
* sunt uşor de partajat datele: tot spaţiul virtual de adrese al unui proces este partajat între thread-urile aceluiaşi proces
* thread-urile se creează mai uşor (se creează o structură internă) şi au overhead de memorie redus (nu se creează o un nou spaţiu de adrese, deci nu se creează o nouă tabelă de pagini)
* schimbarea de context are penalizare mai mică atunci când se schimbă thread-uri ale aceluiaşi proces
+ demo timp de creare thread-uri şi procese
în general thread-urile măresc productivatea/throughput-ul sistemului: mai multă procesare şi mai multe rezultate pe unitatea de timp

thread-urile simplifică programarea atunci când e nevoie de date partajate: nu e nevoie de un API dedicat pentru memorie partajată
folosirea thread-urilor are dezavantaje:
* izolare: partajează spaţiul virtual de adrese şi o problemă a unui thread corupe întreg spaţiul de adresă; un defect (posibilă vulnerabilitate) afectează toate thread-urile
* sincronizare: partajarea spaţiului de adrese poate duce la corupere de date dacă un thread foloseşte datele altui thread
* depanare dificilă: o problemă de programare va fi mai greu identificată într-un program multithreaded: poate fi o problemă cauzată de un thread în memoria altui thread; cu atât mai dificil este când se folosesc biblioteci externe (cod neimplementat de programator)
în cazul unui procesor de video, este recomandat să avem o implementare multi-threaded, care va partaja datele prelucrate şi va folosi mai multe core-uri ale sistemului
un server web poate fi multi-proces sau multi-threaded după cum e mai mult accentul pus pe eficienţă/throughput sau pe securitate/izolare

Threads-uri vs. procese. Atribute ale unui thread

spunem că un proces abstractizează memoria prin spaţiul virtual de adrese, I/O prin descriptori de fişiere şi procesorul prin thread-uri; un procesor poate avea unul sau mai multe thread-uri
un thread abstractizează doar procesorul
un thread este pornit în acelaşi spaţiu de adrese indicându-i-se o funcţie de la care începe execuţia
un proces este pornit dintr-un executabil (exec) sau din procesul curent din punctul curent (fork)
un thread este abstractizarea cea mai simplă a contexului de execuţie; un thread definit simplist printr-un instruction pointer (ce are de executat) şi un stack pointer (o stivă, ce a executat până acum)
schimbarea de context presupune schimbarea unui thread (instruction pointer, stack pointer, celelalte registre) cu alt thread
dacă thread-urile aparţin unor procese diferite e nevoie de schimbarea spaţiului de adrese (schimbare de tabelă de pagini, flush la TLB)
un thread este definit de un TCB (Thread Control Block) conţinând în general informaţii de execuţie: thread id (TID), stare, stivă, pointer la proces / spaţiul de adrese din care face parte, timp de rulare, prioritate
un proces are un thread principal (main); exprimarea "procesul execută o acţiune/instrucţiune" este improprie; spunem că "un thread al procesului (sau thread-ul principal) execută o acţiune/instrucţiune"

Crearea şi încheierea unui thread

când un thread este creat i se indică funcţia ce o va rula
la crearea unui thread i se alocă o stivă şi se începe rularea acelei funcţii
spunem că un thread se activează; e vorba de activarea unui thread
+ demo cu creare de thread-uri şi spaţiu de adresă

thread-ul devine o entitate planificabilă (context de execuţie); se poate bloca, îi expiră cuanta
un thread îşi încheie execuţia:
* când se încheie funcţia
* când se încheie procesul curent (un thread al său apelează exit() sau apare o eroare/excepţie)
* când se apelează o funcţie specifică de închidere a thread-ului (pthread_exit())
similar proceselor există un apel care aşteaptă încheierea unui thread pentru a recupera informaţiile de ieşire; operaţia se numeşte join
un thread poate întoarce o valoare la încheierea sa, recuperată cu join; aşa cum procesul întoarce un cod de ieşire, recuperat cu wait

Partajarea spaţiului de adrese între thread-uri

thread-urile partajează spaţiul de adrese al procesului
spunem că thread-urile au memoria partajată
când un proces nou este creat cu fork() acesta are o copie a tabelei de pagini care referă acelaşi spaţiu de adrese, dar se aplică copy-on-write la acces de scriere
când un thread nou este creat, tot spaţiul de adrese este partajat: o modificare făcută de un thread e vizibilă în alt thread
+ demo cu partajare informaţie între procese şi thread-uri
spunem că thread-urile nu partajează: stiva, o zonă dedicată numită TLS (thread local storage) sau thread specific data
afirmaţia de mai sus este improprie: un thread are acces la stiva altui thread dacă foloseşte construcţii care permit asta; sistemul de operare nu (poate) face enforce ca un thread să nu modifice stiva altui thread; e vorba de acelaşi spaţiu de adrese; similar şi pentru TLS
+ demo cu stiva unui thread în spaţiu de adrese
un thread are stivă proprie, deci variabilele locale sunt proprii fiecărui thread
dacă are nevoie de variabile globale proprii (ca să nu transmită parametri între funcţii) un thread foloseşte TLS; TLS este o zonă dedicată unde fiecare thread are o referinţă a acelei variabile
cel mai simplu mod de a aloca o variabilă în TLS este folosirea atributului __thread în API-ul POSIX
+ demo cu TLS/__thread
pentru accesul coerent la date partajate este nevoie de folosirea de primitive de sincronizare (mai multe în cursul 9: Sincronizare)

Implementarea thread-urilor

API-ul de lucru cu thread-uri (politica) include:
* funcţie de crearea unui thread
* funcţie de aşteptare/join a unui thread
* funcţie de identificare a unui thread (aflare TID)
* funcţie de închidere a unui thread
* primitive de sincronizare

implementarea din spate şi partea de planificare a thread-urilor (mecanismul) este independentă de API
implementarea poate fi: kernel-level threads şi user-level threads (sau green threads, fibers)
kernel-level threads înseamnă suport la nivelul sistemului de operare; thread-urile sunt entităţile planificabile la nivelul sistemului de operare; sunt folosite de planificatorul sistemului de operare, au cuantă de execuţie, stare de execuţie, sunt parte din coada READY
kernel-level threads pot folosi procesoarele sistemului: mai multe thread-uri ale aceluiaşi proces pot rula simultan pe un sistem multi-procesor
schimbarea de context necesită intervenţia nucleului; e nevoie de suport în kernel
user-level threads sunt o implementare completă în user-space; nu este nevoie de suport în kernel
planificatorul este implementat în user-space
o operaţie blocantă a unui user-level thread blochează tot procesul; soluţia este să se folosească operaţii I/O asincrone
se mai numesc green thread în implementările de maşini virtuale (de exemplu JVM)
fibrele sunt thread-uri user space folosite cu planificare cooperativă (yield)
avantaje kernel-level threads: suport complet multi-procesor, bune pentru acţiuni CPU intensive
avantaje user-level threads: nu necesită suport în kernel, timp de activare scurt, performanţă ridicată pentru acţiuni I/O intensive
pentru a reduce timpul de activare se folosesc modele de tipul thread-pool (precum boss-workers sau worker-threads de la APD)

Internele implementării thread-urilor

un thread este implementat ca o structură TCB (thread control block)
o posibilă implementare este o listă de TCB-uri referită de PCB (structura procesului); aşa este în Windows: EPROCESS are o listă de ETHREADS
o altă implementare este o structură pentru address space şi thread-uri care pot fi ataşate de adress space; nu există structură de proces efectivă; se întâmplă la microkernel-ul L4
în Linux un thread sau un proces sunt reprezentate de structura task_struct; dacă două structuri partajează spaţiul de adrese spunem că sunt thread-uri ale aceluiaşi proces
în user-space implementarea este specifică bibliotecii de thread-uri sau maşinii virtuale

Sumar

acţiunile în sistemul de calcul sunt de lucru cu I/O şi prelucrare: I/O intensive şi CPU intensive
în mod tradiţional, acţiunile sunt încapsulate în procese
pe un sistem multi-core putem executa mai multe acţiuni (procese) diferite sau un proces clonat şi cu memorie partajată între el şi procesele copil
dezavantajele proceselor sunt overhead de creare, planificare şi consum de memorie (heavyweight)

thread-urile sunt lightweight processes cu timp de creare şi rulare (activare) rapid, partajează spaţiul de adresă
thread-urile au probleme din cauza lipsei de izolare între ele: corupere de memorie, coruperea întregului proces
un thread abstractizează procesorul: stivă/stack pointer şi instruction pointer
un thread e definit de un TCB (thread control block)
activarea unui thread se face prin rularea unei funcţii specifice thread-ului
întreg spaţiul de adresă este partajat între thread-uri ale aceluiaşi proces; TLS (Thread Local Storage) şi stiva sunt specifice fiecărui thread, dar pot fi în continuare accesate de un alt thread
thread-urile sunt implementate în forma kernel-level threads sau user-level threads
kernel-level threads au suport complet la nivelul nucleului, pot folosi suportul multiprocesor
user-level threads nu au suport în kernel (sunt implementate în user space), dar au timp de activare mai rapid