Sisteme de operare

26 iunie 2009

Timp de lucru: 70 de minute

NOTĂ: toate răspunsurile trebuie **justificate**

1. Știind că operațiile de lucru cu pipe-uri sunt atomice, implementați în pseudocod un mutex cu ajutorul pipe-urilor.

lock: read(pipefd[0], &a, 1);
unlock: write(pipefd[1], &a, 1);

a este un char; pipefd este un pipe

2. Durata unei schimbări de context este de 1ms iar overhead-ul unui apel de sistem de 100µs. Totuși, la un moment dat, apelul down(&sem); durează doar 1µs. Apelul se realizează cu succes. Care este explicația?

Apelul down este implementat în user-space (de exemplu o implementare de tip futex). Dacă valoarea semaforului este strict pozitivă, atunci apelul down va decrementa valoarea semaforului și va continua execuția (fără apel de sistem și fără schimbare de context). În cazul în care valoarea este egală cu 0 va avea loc un context switch. În cazul unei implementări de thread-uri kernel-level, acest lucru va presupune și un apel de sistem (planificatorul este implementat în kernel-space).

3. De ce este mai avantajos ca, pe un sistem uniprocesor, după un apel *fork* să fie planificat primul procesul fiu?

Pentru a evita posibilele copieri inutile datorate copy-on-write. De multe ori, după fork procesul copil execută exec, rezultând în schimbarea completă a spațiului de adresă. Dacă procesul părinte ar fi planificat primul, atunci apelurile de scriere ale acestuia vor rezulta în duplicarea paginilor (overhead temporal și consum memorie) datorită copy-on-write. Dacă procesul copil face exec, atunci acele duplicate au însemnat un consum inutil de resurse.

4. Există vreo diferență între implementarea simbolului *errno* în contextul unui proces single-threaded față de un proces multi-threaded? Argumentați.

Da, există diferență. Fiecare thread trebuie să aibă acces la o variabilă errno proprie, astfel că errno va fi de obicei implementat ca variabilă per-thread. Acest lucru se poate realiza cu ajutorul TLS/TSD. O variabilă comună errno pentru toate thread-uri ar conduce la incosistența informațiilor referitoare la erorile apărute.

5. Un sistem uniprocesor (single-core) dispune de 64KB L1 cache, 512KB L2 cache și un TLB cu 256 intrări. Pe un sistem de operare cu suport în kernel pentru thread-uri, ce durează mai mult: schimbarea de context între două thread-uri sau între două procese?

Schimbarea de context între două procese va dura tot timpul mai mult decât schimbarea de context între două procese. În momentul schimbării de context între două procese se schimbă întreg spațiul de adresă și resursele asociate. Se schimbă astfel tabela de pagini, se face flush la TLB etc. În cazul thread-urilor o schimbare de context presupune doar schimbarea registrelor și a informațiilor specifice unui thread.

6. Comanda pmap afișează informații despre spațiul de adrese al unui proces. În urma rulării de mai jos a comenzii *pmap* se observă următoarele informații despre biblioteca standard C:

```
# pmap 1
base address          size    rights     name
[...]
00007f8c480e6000    1320K     r-x--      libc-2.7.so
00007f8c4842f000      12K     r----      libc-2.7.so
00007f8c48432000       8K     rw---      libc-2.7.so
```

Presupunând că în sistem rulează 50 de procese care folosesc biblioteca standard C, care este spațiul total de memorie RAM ocupat de bibliotecă?

*Ultima zona este o zonă read-write și nu poate fi partajată între două procese. Celelalte două zone sunt read-only și vor fi partajate. Biblioteca va ocupa, așadar, 50*8K + 12K + 1320K.*

7. Descrieți și explicați în pseudo-asamblare cum acționează suportul de SSP (Stack Smashing Protection) pe un sistem în care stiva crește în sus (de la adrese mici la adrese mari).

Pe un sistem pe care stiva crește în sus nu se poate realiza stack overflow din stack-frame-ul curent ci din stack frame-ul apelantului. Astfel, dacă o funcție apelează strcpy și un argument este un buffer al funcției, acest buffer poate fi folosit pentru a suprascrie (prin overflow) adresa de retur a funcției strcpy. Valoarea de tip canary trebuie stocată la o adresă mai mică decât adresa de retur a funcției strcpy (practic, la fel ca la o stivă care crește în jos). Întrucât apelantul este cel care construiește stack frame-ul apelatului, acesta va trebui să marcheze valoarea de tip canary. În schimb apelatul (aici strcpy), înainte de întoarcere va verifica suprascrierea adresei de tip canary.
Stack frame-ul este cel de mai jos:

```
[ strcpy local  ]
[      ...      ]
[ ret address   ]
[ old_ebp       ]  callee (strcpy) stack frame
[ canary value  ]----
[ strcpy param1 ]
[ strcpy param2 ]
[      ...      ]
[ local buffer  ]  caller stack frame
[      ...      ]
[ local buffer  ]
```

8. Pe un sistem de fișiere dat un dentry are următoarea structură:
- 1 octet - lungimea numelui
- 251 octeți – numele
- 4 octeți - numărul inode-ului

O instanță a unui astfel de sistem de fișiere deține un director rădăcină, 5 subdirectoare, iar fiecare subdirector conține 5 fișiere. Câte dentry-uri deține sistemul de fișiere?

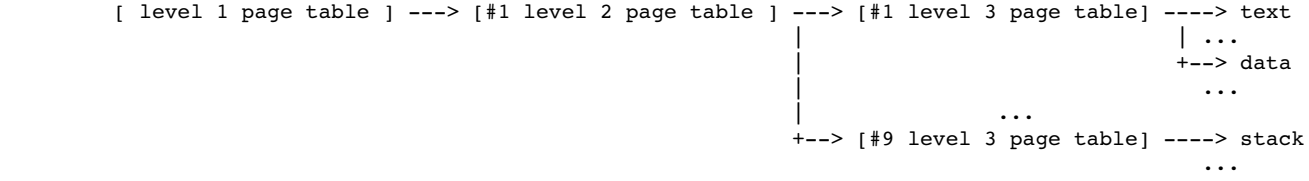
*Fiecare intrare în sistemul de fișiere (director sau fișier) conține cel puțin un dentry. Ignorând intrările speciale . și .. rezultă (1 +) 5 + 5*5 = 30 (31) intrări. Directorul rădăcină poate să nu aibă dentry. Considerând intrările speciale, rezultă un plus de 1 + 2*5 intrări = 11 intrări (directorul rădăcină nu are referință ..).*

9. Un sistem pe 64 de biți folosește pagini de 8KB și 43 de biți pentru adresare într-o schemă de adresare ierarhică pe trei niveluri cu împărțirea (10 + 10 + 10 + 13). Un proces care rulează în cadrul acestui sistem are, la un moment dat, următoarea componență a spațiului de adrese (se începe de la adresa 0):

```
[ text ]             - 16 pagini
[ data ]             - 8 pagini
[ spațiu nealocat ]  - 8168 pagini
[ stivă]             - 8 pagini
```

Știind că o intrare în tabela de pagini ocupă 64 de biți, câte pagini ocupă tabelele de pagini pentru procesul dat?

O intrare în tabela de pagini ocupă 64 de biți = 8 octeți. Există, astfel, 1024 de intrări într-o pagină. Zona text și data ocupă 24 de pagini deci vor există 24 de intrări valide în prima pagină de tabelă de pe nivelul 3. Următoarele 8168 pagini vor completa intrările din prima tabelă de pe nivelul 3 și vor mai folosi 7 pagini de tabele. Întrucât este spațiu nealocat, cele 7 pagini de tabele nu vor fi nici ele alocate. A 9-a pagină de tabela va folosi primele 8 intrări pentru a referi paginile de pe stivă.
Prima pagină de tabelă de pe nivelul 2 va avea valide doar prima și a 9-a intrare (care vor referi prima și a 9-a pagină de tabelă). Pagina de tabelă de pe nivelul 1 va avea validă doar prima intrare către pagina de tabelă de pe nivelul 2. Vor fi, astfel, folosite, doar 4 pagini.
Schematic, reprezentarea este următoarea:



10. Dați exemplu de situație în care, pentru comunicația cu dispozitivele de I/E, se preferă folosirea polling în loc de întreruperi.

Pollingul se preferă în situațiile în care întreruperile previn funcționarea eficientă a sistemului. Acest lucru se întâmplă în cazul în care întreruperile sunt transmise foarte des și procesorul petrece mult timp în rutinele de tratare a întreruperilor. Soluția este dezactivarea temporară a întreruperilor și folosirea polling. Acest lucru se întâmplă la dispozitivele de rețea foarte rapide, spre exemplu plăcile de rețea.

11. Un program execută secvența de cod din coloana din stânga tabelului de mai jos. În coloana din dreapta este prezentat rezultatul rulării programului:

```
/* init array to 2, 0, 0, 0 ... */
static int data1[1024*1024] = {2, };

static void print_time(char *msg)
// ...

static void init_array(int *a, size_t len)
{
    size_t i;

    for (i = 0; i < len; i += 1024)
        a[i] = 2009;
}

int main(void)
{
    int *data2 = malloc(1024*1024 * sizeof(int));

    print_time("before init data1");
    init_array(data1, 1024*1024);
    print_time("after  init data1");

    print_time("before init data2");
    init_array(data2, 1024*1024);
    print_time("after  init data2");

    return 0;
}
```

```
before init data1: 1245582962s, 753431us
after  init data1: 1245582962s, 767496us

before init data2: 1245582962s, 767524us
after  init data2: 1245582962s, 776012us




---
Se observa ca:
-   durata initializare data1 - 14065us
-   durata initializare data2 -  8488us
```

Cum explicați faptul că inițializarea vectorului *data1* durează mai mult decât inițializarea vectorului *data2*?

Data1 se găsește în .data și este stocat în executabil. Zona .data a executabilului va fi mapată în memorie folosind demand paging. Drept consecință, un page-fault în momentul inițializării vectorului data1 va forța citirea de pe disc (din executabil). De partea cealaltă, data2 va fi alocat direct în RAM la cerere (tot prin demand-paging).