1. Două procese (P1, P2) folosesc o zonă de memorie partajată, rulează într-un sistem preemptiv și execută secvența de cod de mai jos. Știind că valoarea inițială indicată de pointerul **counter** este 0 și că acesta indică în zona de memorie partajată, care este valoarea indicată de pointerul **counter** la finalul execuției celor două procese?

P1	P2
`if (*counter == 0)` `  (*counter)++;`	`if (*counter == 0)` `  (*counter)++;`

Dacă cele două procese se execută secvențial atunci primul proces va incrementa variabila pe 1, iar al doilea nu va executa instrucțiunea de incrementare din if. Valoarea finală va fi 1.

*În varianta în care procesul P1 este preemptat de procesul P1 după verificarea if (*counter == 0), dar înainte de (*counter)++ atunci procesul P2 va testa, la rândul său, ca fiind adevărată condiția (*counter == 0). În consecință, va incrementa valoarea contorului. P1 va incrementa, de asemenea, valoarea contorului, rezultând valoarea finală 2.*

2. Pe un sistem dual processor cu 128MB de RAM și swap de 256MB rulează un sistem Linux. Câte procese se pot găsi la un moment dat în starea **RUNNING**, **READY** respectiv **WAITING**?

*Caracteristica ce influențează numărul de procese din starea RUNNING este numărul de unități de execuție. Fiind vorba de un sistem dual procesor, pot exista maxim **2 procese în coada RUNNING**.*
***În coada READY și WAITING se pot găsi oricâte procese**, limita fiind dată de constrângerile și resursele sistemului.*

3. Un sistem folosește TLB în care fiecare intrare conține trei câmpuri (pid, frame, pagină virtuală). Care este avantajul acestei implementări față de o implementare care conține doar două câmpuri (frame, pagină virtuală)?

Prezența câmpului pid înseamnă că se poate realiza o selecție după proces a intrărilor din TLB. Acest lucru este util în cazul schimbărilor de context. În cazul unei schimbări de context cea mai mare parte a intrările din TLB sunt anulate. Prezența unui câmp pid înseamnă că, în cazul unei schimbări de context, doar intrările specifice procesului vor fi eliminate rezultând într-un număr mai mic de accese directe la tabela de pagini.

4. Descrieți în pseudocod cum se poate determina dacă stiva crește de la adrese mari la adrese mici sau invers.

O soluție ține cont de construirea stack frame-urilor pentru funcții:

```
int *p_fcaller;
int *p_fcalled;

void f2(void)
{
        int f2_local;
        p_fcalled = &f2_local;

        if (p_fcalled > p_fcaller)
                printf("stack grows up\n");
        else
                printf("stack grows down\n");
}
void f1(void)
{
        int f1_local;
        p_fcaller = &f1_local;
        f2();
}
```

Exemplu complet aici: http://swarm.cs.pub.ro/git/?p=razvan-code.git;a=blob;f=tests/stack_grow.c;h=3ebc0408a4cf30cb8317714fb5c473c35ca7bb3d;hb=21d876d26d3db2145f6be96423cbc14a7ddaf21a

5. Câte pagini fizice vor ocupa stivele celor două procese rezultate în urma apelului **fork** exact înainte de apelul **exit**? Apelul **fork** se întoarce cu succes.

```
int main(void)
{
        char buf[3 * PAGE_SIZE];
        buf[0] = 'a';
        buf[PAGE_SIZE] = 'z';
        fork();
        buf[0] = 'b';
        exit(EXIT_SUCCESS);
}
```

Bufferul buf este alocat folosind demand paging. Acest lucru înseamnă că nu vor fi alocate pagini fizice până în momentul accesului. După buf[0] = 'a' și buf[PAGE_SIZE] = 'z' rezultă două page fault-uri care vor genera alocare a două pagini fizice.

După fork paginile sunt marcate copy-on-write.

După apelul fork() atât procesul părinte cât și procesul copil accesează buf[0]. Acest lucru va rezulta într-un page fault (se duplică pagina și pagina nouă si pagina veche primesc drept de scriere). Se va aloca astfel o pagină fizică nouă.

În final stivele celor două procese vor ocupa 3 pagini fizice.

6. În urma rulării secvenței de cod de mai jos fișierul **a.txt** conține șirul **222313**. După fiecare apel **write** actualizați următorul vector (cursor fd1, cursor fd2, cursor fd3, conținut fișier). Exemplu: inainte de primul write vectorul va fi (0, 0, 0, '').

```
fd1 = open("a.txt", O_RDWR | O_CREAT | O_TRUNC, 0644);
fd2 = open("a.txt", O_RDWR | O_CREAT | O_TRUNC, 0644);
fd3 = dup(fd1);

write(fd1, "1", 1);
write(fd2, "2", 1);
write(fd3, "3", 1);

pid = fork();


/* continuat pe coloana a doua */
```

```
switch (pid) {
case 0:
        write(fd1, "1", 1);
        write(fd2, "2", 1);
        write(fd3, "3", 1);
        break;
default:
        wait(&status);
        write(fd1, "1", 1);
        write(fd2, "2", 1);
        write(fd3, "3", 1);
        break;
}
```

	inițial	w1	w2	w3	w4	w5	w6	w7	w8	w9
f1.cursor	0	1	1	2	3	3	4	5	5	6
f2.cursor	0	0	1	1	1	2	2	2	3	3
f3.cursor	0	1	1	2	3	3	4	5	5	6
conținut	“”	“1”	“2”	“23”	“231”	“221”	“2213”	“22131”	“22231”	“222313”

Procesul părinte și fiu partajează descriptorii de fișier și cursorul de fișier. Orice modificare a cursorului în procesul copil va fi vizibilă în procesul părinte și invers.
fd3 partajează cursorul de fișier cu fd1 ca urmare a apelului dup. Orice incrementare a cursorului folosind descriptorul fd1 va fi vizibila descriptorului fd3 și invers.