1. Fie următoarele procese și timpii lor de execuție și de intrare (process, timp execuție, timp intrare): (P1, 8, 1), (P2, 4, 2), (P3, 3, 3), (P4, 7, 4). Determinați timpii medii de așteptare pentru First Come First Served (FCFS) și Shortest Job First (SJF).

În cadrul răspunsului, am marcat cu (a, b) intervalul în care un proces așteaptă și cu [a, b] intervalul în care un proces este activ pe procesor

= FCFS =

P1: [1,9]
P2: (2, 9) [9,13]
P3: (3, 13) [13,16]
P4: (4, 16) [16,23]

Procesul P1 intră în sistem la momentul 1 și, fiind primul proces, lucrează 8 unități de timp până la momentul 9. Nu așteaptă.
Procesul P2 intră în sistem la momentul 2 și așteaptă procesul P1; va fi planificat înaintea proceselor P3 și P4. Va aștepta până la încheierea procesului P1 adică intervalul (2, 9) Începe să lucreze în intervalul [9,13] pentru 4 unități de timp.
Procesul P3 intră în sistemul la momentul 3 și așteaptă procesul P1 și P2, adică intervalul (3,13). Va lucra în intervalul [13,16] pentru 3 unități de timp.
Procesul P4 intră în sistemul la momentul 4 și așteaptă procesul P1, P2 și P3, adică intervalul (4,16). Va lucra în intervalul [16,23] pentru 7 unități de timp.

Timp mediu de așteptare = (0 + 7 + 10 + 12) / 4 = 7.25

= SJF =

P1: [1,9]
P2: (2,12)[12,16]
P3: (3,9)[9,12]
P4: (4,16)[16,23]

** Procesul P1 întră în sistem la momentul 1 și rulează pe procesor [1,9].*
** La momentul 2 intră procesul P2 al cărui timp de execuție este 4. Întrucât primul proces rulează, nu va preempta procesorul.*
** La momentul 3 intră procesul P3 al cărui timp de execuție este 3. Întrucât primul proces rulează, nu va preempta.*
** La momentul 4 intră procesul P4 al cărui timp de execuție este 7. Nu preemptează nici un proces și așteaptă.*
** La momentul 9, P1 își încheie execuția. Se planifică procesul cu timpul de execuție cel mai scurt. Se planifică P3 (timp de execuție 3). P3 rulează pe procesor în intervalul [9,12].*
** La momentul 12, P3 își încheie execuția. Se planifică procesul cu timpul de execuție cel mai scurt. Se planifică P2 (timp de execuție 4). P3 rulează pe procesor în intervalul [12,16].*
** La momenutl 16, P2 își încheie execuția. Se rulează procesul P4.*
** La momentul 23, procesul P4 își încheie execuția.*

Timp mediu de așteptare = (0 + 10 + 6 + 12) / 4 = 7

2. Câte procese se vor crea în urma execuției secvenței de mai jos (se exclude procesul curent)? Toate apelurile fork se întorc cu succes.

```
l1)    fork();
l2)    if (fork() == 0)
l3)            fork();
```

Apelul fork() întoarce 0 în procesul copil și cu PID-ul procesului copil în părinte
Fie P1 primul proces (cel existent).
După linia 1 (l1) primul proces creează un proces copil, P2. Întrucât nu se verifică valoarea de ieșire a apelului fork, ambele procese (P1 și P2) vor continua cu linia l2
În cadul liniei 2 cele două procese de mai sus creează câte un proces copil. Fie P3 procesul copil pentru P1 și P4 procesul copil pentru P2. În cadrul liniei se verifică valoarea întoarsă de apelul fork. Doar procesele copil proaspăt create vor satisface condiția if (fork() == 0). Adică foar procesele P3 și P4 vor continua la linia l3.
În cadrul liniei l3, procesele P3 respectiv P4 creează câte un proces copil; fie acestea P5 și P6.
La sfârșitul secțiunii vor exista 6 procese: procesul inițial (P1) și 5 noi procese create (P2, P3, P4, P5, P6).

3. Fie secvențele de pseudocod de mai jos. Care din cele două abordări este optimă?

`mutex_lock(&mutex);`
`a++;`
`mutex_unlock(&mutex);`
	`spin_lock(&spinlock);`
	`a++;`
	`spin_unlock(&spinlock);`

Secvențele de mai sus urmăresc accesul exclusiv pentru incrementarea variabilei a. Incrementarea variabilei a este o operație rapidă și puțin consumatoare de timp. Se dorește, așadar, un mecanism de sincronizare/asigurare a accesului exclusiv cât mai rapid. Acest mecansim este asigurat de spinlock-uri care operează foarte rapid în cadrul primitivelor spin_lock și spin_unlock. În vreme ce o operație pe un mutex impune contactarea planificatorului, spinlock-ul impune o operație de busywaiting care va fi, însă, foarte rapidă ținând cont de dimensiunea redusă a regiunii critice (necesită doar incrementarea variabilei a).

4. În secvența de cod de mai jos, PAGE_SIZE reprezintă dimensiunea unei pagini, iar apelul mmap reușește. În urma rulării a 10 instanțe ale programului de mai jos se ocupă 11 pagini fizice în zonele de date (date inițializate, heap, readonly (.rodata), bss). Argumentați acest comportament.

```
const char x[PAGE_SIZE] = {1, };

int main(void)
{
        char *z = mmap(NULL,  PAGE_SIZE, PROT_READ, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
        printf("%c %c\n", x[0], z[0]);
        while (1)  /* wait forever */
                ;
        return 0;
}
```

Vectorul x ocupă o pagină și fiind declarat const este read-only. Acest lucru înseamnă că pagina fizică aferentă acestuia poate fi partajată între mai multe procese. Nu este nevoie de copie separată pentru fiecare proces în parte.

Vectoul z ocupă, de asemnea, o pagină. Fiind vorba de o mapare privată fiecare proces va dispune de o pagină separată.

Alocările se realizează folosind demand-paging. Doar în momentul accesului la date, se vor aloca pagini fizice.

*Pentru fiecare instanță de proces, se alocă, în momentul accesului z[0] o pagină fizică nouă pentru z. Pagina fizică aferenta lui x va fi identică pentru toate procesele, întrucât este read-only. După rularea a 10 procese vor rezulta 10 * 1 pagină (pentru z) + 1 pagină (pentru x) = 11 pagini fizice.*

5. Care dintre apelurile sigaction și sigemptyset va dura mai mult? sigaction actualizează rutina de tratare a unui semnal, iar sigemptyset initializeaza setul de semnale descris de structura sigset_t la 0.

sigaction este apel de sistem și are un overhead inerent. sigemptyset operează la nivelul biților dintr-o zonă de memorie, operație foarte rapidă și care nu dispune de un overhead suplimentar. În consecință, durata sigaction este vizibil mai mare decât durata sigemptyset. Puteti verifica pe codul de aici: http://swarm.cs.pub.ro/git/?p=razvan-code.git;a=blob;f=tests/measure_time_syscall.c;h=f7046aedb4b8af9ffe65cc84533ca43c3888a1d6;hb=refs/heads/examples

6. Fie un sistem pe 32 biți cu 100MB RAM, 100MB de swap și pagini de 4KB. Spațiul de adresă virtual este împărțit 3GB programe utilizator / 1 GB kernel. Un program aflat în execuție rulează:

```
        void *ptr = malloc(1024 * 1024 * 1024);
```

Apelul se realizează cu success, returnând un pointer valid. Motivați de ce malloc se întoarce cu succes.
Ce se întâmplă și de ce, dacă se rulează:

```
        memset(ptr, 0, 1024 * 1024 * 1024);
```

malloc alocă memorie folosind demand-paging. În concluzie se alocă memorie pur virtuală fără a dispune de suport fizic. Memoria fizică nu este folosită și apelul reușește.

În cadrul apelului memset, se produc page-fault-uri la accesarea diferitelor adrese indicate de ptr. Pentru fiecare page-fault se alocă o nouă pagină fizică. În jurul valorii de 200MB, spațiul din memoria RAM și cel de pe swap se vor fi umplut, drept pentru care sistemul va rămâne fără memorie, se va activa un sistem de management de tipul OOM (Out of memory) (http://en.wikipedia.org/wiki/Out_of_memory) care va începe să omoare procese. În final, sistemul va suferi crash/se va bloca din cauza absenței memoriei fizice.