Fundamentos de los Sistemas Operativos (FSO)

Departamento de Informática de Sistemas y Computadoras (DISCA)
Universitat Politècnica de València

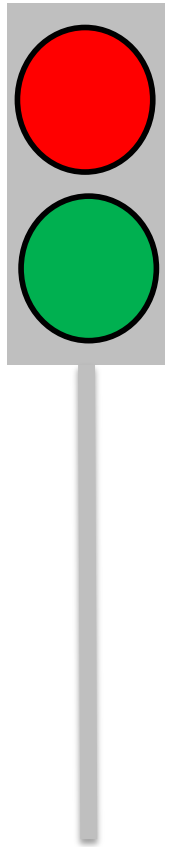
Part 2: Process management

Seminar 6

Synchronization: POSIX Semaphores

fSO

- OS level solutions
- POSIX semaphores
- POSIX mutexes
- **Exercises**



- **Exercise S06.1** What possible values will take **x** as a result of the concurrent execution of the following threads?

```
#include <semaphore.h>
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
sem_t s1,s2,s3;
int x;
```

```
void *func_thread1(void *a)
{
 sem_wait(&s1);
 sem_wait(&s2);
 x=x+1;
 sem_post(&s3);
 sem_post(&s1);
 sem_post(&s2);
}
void *func_thread2(void *b)
{
 sem_wait(&s2);
 sem_wait(&s1);
 sem_wait(&s3);
 x=10*x;
 sem_post(&s2);
 sem_post(&s1);
}
```

```
int main()
{
  pthread_t h1,h2 ;
  x = 1;
  sem_init(&s1,0,1); /*Inicializa a 1*/
  sem_init(&s2,0,1); /*Inicializa a 1*/
  sem_init(&s3,0,0); /*Inicializa a 0*/

  pthread_create(&h1,NULL,func_thread1,NULL);
  pthread_create(&h2,NULL,func_thread2,NULL);
  pthread_join(h1,NULL);
  pthread_join(h2,NULL);
}
```

- **Exercise S06.1** What possible values will take **x** as a result of the concurrent execution of the following threads?

```
#include <semaphore.h>
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
sem_t s1,s2,s3;
int x;
```

```
void *func_thread1(void *a)
{
 sem_wait(&s1);
 sem_wait(&s2);
 x=x+1;
 sem_post(&s3);
 sem_post(&s1);
 sem_post(&s2);
}
void *func_thread2(void *b)
{
 sem_wait(&s2);
 sem_wait(&s1);
 sem_wait(&s3);
 x=10*x;
 sem_post(&s2);
 sem_post(&s1);
}
```

```
int main()
{
  pthread_t h1,h2 ;
  x = 1;
  sem_init(&s1,0,1); /*Inicializa a 1*/
  sem_init(&s2,0,1); /*Inicializa a 1*/
  sem_init(&s3,0,0); /*Inicializa a 0*/

  pthread_create(&h1,NULL,func_thread1,NULL);
  pthread_create(&h2,NULL,func_thread2,NULL);
  pthread_join(h1,NULL);
  pthread_join(h2,NULL);
}
```

Initial state

s1	1
s2	1
s3	0

x = 1

Scenario 1

If h1 arrives first:

s1	0
s2	0
s3	0

x = 2

s1	1
s2	1
s3	1

If h2 arrives second:

s1	0
s2	0
s3	0

x = 20

s1	1
s2	1
s3	0

- **Exercise S06.1** What possible values will take **x** as a result of the concurrent execution of the following threads?

```
#include <semaphore.h>
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
sem_t s1,s2,s3;
int x;
```

s1	1
s2	1
s3	0

Initial state

x = 1

```
void *func_thread1(void *a)
{
 sem_wait(&s1);
 sem_wait(&s2);
 x=x+1;
 sem_post(&s3);
 sem_post(&s1);
 sem_post(&s2);
}
void *func_thread2(void *b)
{
 sem_wait(&s2);
 sem_wait(&s1);
 sem_wait(&s3);
 x=10*x;
 sem_post(&s2);
 sem_post(&s1);
}
```

```
int main()
{
  pthread_t h1,h2 ;
  x = 1;
  sem_init(&s1,0,1); /*Inicializa a 1*/
  sem_init(&s2,0,1); /*Inicializa a 1*/
  sem_init(&s3,0,0); /*Inicializa a 0*/

  pthread_create(&h1,NULL,func_thread1,NULL);
  pthread_create(&h2,NULL,func_thread2,NULL);
  pthread_join(h1,NULL);
  pthread_join(h2,NULL);
}
```

Scenario 2

If h2 arrives first:

s1	0
s2	0
s3	0

blocked

If h1 arrives later:

s1	0	blocked
s2	0	
s3	0	

- **Exercise S06.1** What possible values will take **x** as a result of the concurrent execution of the following threads?

```
#include <semaphore.h>
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
sem_t s1,s2,s3;
int x;
```

s1	1
s2	1
s3	0

Initial state

x = 1

```
void *func_thread1(void *a)
{
 sem_wait(&s1);
 sem_wait(&s2);
 x=x+1;
 sem_post(&s3);
 sem_post(&s1);
 sem_post(&s2);
}
void *func_thread2(void *b)
{
 sem_wait(&s2);
 sem_wait(&s1);
 sem_wait(&s3);
 x=10*x;
 sem_post(&s2);
 sem_post(&s1);
}
```

```
int main()
{
  pthread_t h1,h2 ;
  x = 1;
  sem_init(&s1,0,1); /*Inicializa a 1*/
  sem_init(&s2,0,1); /*Inicializa a 1*/
  sem_init(&s3,0,0); /*Inicializa a 0*/

  pthread_create(&h1,NULL,func_thread1,NULL);
  pthread_create(&h2,NULL,func_thread2,NULL);
  pthread_join(h1,NULL);
  pthread_join(h2,NULL);
}
```

Scenario 3

If h1 arrives first and is preempted after s1 wait

s1	0
s2	1
s3	0

then h2 executes s2 wait and it wait on s1 that is 0 => blocked

s1	0
s2	0
s3	0

then h1 is executed again wait on s2 => blocked

s1	0
s2	0
s3	0

Exercise S06.2 What possible values will take shared variables **x** and **y** at the end of the following concurrent threads. The initial values are: x=1, y=4, S1=1, S2=0 y S3=1.

Thread A	**Thread B**	**Thread C**
P(S2);	P(S1);	P(S1);
P(S3);	P(S3);	P(S3);
x = y * 2;	x = x + 1;	x = y + 2;
y = y + 1;	y = 8 + x;	y = x * 4;
V(S3);	V(S2);	V(S3);
	V(S3);	V(S1);

Exercise S06.2 What possible values will take shared variables **x** and **y** at the end of the following concurrent threads. The initial values are: x=1, y=4, S1=1, S2=0 y S3=1.

Thread A	Thread B	Thread C

P(S2);
P(S3);
x = y * 2;
y = y + 1;
V(S3);
	P(S1);
	P(S3);
	x = x + 1;
	y = 8 + x;
	V(S2);
	V(S3);
		P(S1);
		P(S3);
		x = y + 2;
		y = x * 4;
		V(S3);
		V(S1);

Initial state

s1	1
s2	0
s3	1
x	1
y	4

ABCA

A P(S2)

s1	0
s2	0
s3	0
x	1
y	4

B P(S1)P(S3)

s1	0
s2	0
s3	0
x	2
y	10

B V(S2)V(S3)

s1	0
s2	1
s3	1
x	2
y	10

C P(S1) blocked

A P(S3).

s1	0
s2	0
s3	0
x	20
y	11

V(S3)

s1	0
s2	0
s3	1
x	20
y	11

Initial state

s1	1
s2	0
s3	1
x	1
y	4

BCA
BAC

B P(S1)P(S3)

s1	0
s2	0
s3	0
x	1
y	4

s1	0
s2	0
s3	0
x	2
y	10

B V(S2)V(S3)

s1	0
s2	1
s3	1
x	2
y	10

C P(S1) blocked

A P(S2) P(S3)

s1	0
s2	0
s3	0
x	20
y	11

V(S3)

s1	0
s2	0
s3	1
x	20
y	11

Exercise S06.2 What possible values will take shared variables **x** and **y** at the end of the following concurrent threads. The initial values are: x=1, y=4, S1=1, S2=0 y S3=1.

Thread A	**Thread B**	**Thread C**

P(S2);
P(S3);
x = y * 2;
y = y + 1;
V(S3);
	P(S1);
	P(S3);
	x = x + 1;
	y = 8 + x;
	V(S2);
	V(S3);
		P(S1);
		P(S3);
		x = y + 2;
		y = x * 4;
		V(S3);
		V(S1);

Initial state

s1	1
s2	0
s3	1
x	1
y	4

C P(S1) PS(3)

s1	0
s2	0
s3	0
x	6
y	24

V(S3)VS1)

s1	1
s2	0
s3	1
x	6
y	24

CBA

A P(S2)

B P(S1)P(S3)

s1	0	s1	0
s2	0	s2	0
s3	0	s3	0
x	6	x	7
y	24	y	15

B V(S2)V(S3)

s1	0
s2	1
s3	1
x	7
y	15

A P(S3).

s1	0
s2	0
s3	0
x	30
y	16

V(S3)

s1	0
s2	0
s3	1
x	30
y	16

CABA

B P(S1)P(S3)

s1	0	s1	0
s2	0	s2	0
s3	0	s3	0
x	6	x	7
y	24	y	15

B V(S2)V(S3)

s1	0
s2	1
s3	1
x	7
y	15

A P(S2)P(S3).

s1	0
s2	0
s3	0
x	30
y	16

V(S3)

s1	0
s2	0
s3	1
x	30
y	16

Exercise 6.3:

Describe if the following code is a good solution to the critical section problem when there are only two threads involved (thread_0 and thread_1)

```
#include <stdio.h>
/** Shared **/
  int flag[2];
```

```
thread_i(void) {
  while ( 1 ) {

   remaining_section;

    flag[i] = 1;
    while(flag[(i+1) % 2]);

    critical_section;


    flag[i] = 0;
  }
}
```

¿Does it comply with the three critical section protocol conditions?

Exercise 6.3:

Describe if the following code is a good solution to the critical section problem when there are only two threads involved (thread_0 and thread_1)

```
#include <stdio.h>
/** Shared **/
  int flag[2];
```

```
thread_i(void) {
  while ( 1 ) {

   remaining_section;

    flag[i] = 1;
    while(flag[(i+1) % 2]);

    critical_section;

    flag[i] = 0;
  }
}
```

¿Does it comply with the three critical section protocol conditions?

- ***Mutual exclusion:*** if an activity is inside its critical section then no other activities can be at the same time in their critical sections, the others must wait until the critical section is free
- ***Progress:*** if the critical section is free and there are more than one activity willing to enter it, the decision about which one enters is taken in finite time and only depends on the waiting activities
- ***Limited waiting:*** After an activity has asked to enter the critical section there is a limited number of times other activities are allowed entering the critical section

Progress => fail

No progress => deadlock

Scenario: th0 set flag[0] = 1; preempted
th1 set flag[1] = 1; while (flag[0] == 1) ; till flag[0] = 0
th1 while (flag[1] == 1) ; till flag[1] = 0

Exercise 6.2:

Compare the three test-and-set based proposed solutions to the critical section problem

```
/* Solution a */
void *thread(void *p) {

  while(1) {
   while (test_and_set(&key));
   /* Critical section */
   key = 0;
   /* Remaining section */
  }
}
```

```
/* Solution b */
void *thread(void *p) {

 while(1) {
 while (test_and_set(&key)) usleep(1000);
   /* Critical section */
   key = 0;
   /* Remaining section */
  }
}
```

```
#include <stdio.h>
/** Shared **/
  int key=0;
```

```
/* Solution c*/
void *thread(void *p) {

 while(1) {
 while (test_and_set(&key)) yield();
   /* Critical section */
   key = 0;
   /* Remaining section */
  }
}
```

Note. yield() call allows leaving the remaining CPU time to others. In this way the thread that calls to yield() goes to READY and frees the CPU, and gives the scheduler the opportunity to select another activity for execution

Exercise 6.4:

Compare the three test-and-set based proposed solutions to the critical section problem

```
for (i=0; i<(iterations); i++) {
   aux_variable = sharedVar;
   aux_variable++;
   sharedVar = aux_variable;
}

Shared Variable= 40000000
real    0m0,168s
user    0m0,161s
sys     0m0,000s
```

```
for (i=0; i<iter; i++){
   while (__sync_lock_test_and_set(&control, 1)) ;
   .....
   __sync_lock_release(&control);
}

real    0m1,049s
user    0m1,024s
sys     0m0,000s
```

```
for (i=0; i<iter; i++){
   while (__sync_lock_test_and_set(&control, 1))
          sched_yield();
      ..........
   control = 0;
}
real    0m1,048s
user    0m0,634s
sys     0m0,389s
```

```
for (i=0; i<iter; i++){
   while (__sync_lock_test_and_set(&control, 1))
          usleep(1);
      ..........
   control = 0;
}
1 usec                      usleep(10);
real    0m0,255s            real    0m0,253s
user    0m0,248s            user    0m0,246s
sys     0m0,000s

usleep(100);                usleep(1000);
real    0m0,254s            real    0m0,268s
user    0m0,242s.           user    0m0,253s
```

```
for (i=0; i<iter; i++){
   sem_wait(&sem);
   .......
   sem_post(&sem);
}

real    0m2,103s
user    0m1,640s
sys     0m0,426s
```

```
for (i=0; i<iter; i++){
   pthread_mutex_lock(&mutex);
   .....
   pthread_mutex_unlock(&mutex);
}

real    0m0,638s
user    0m0,616s
sys     0m0,000s
```

Exercise 6.3:

The following functions are executed by two concurrent threads inside the same process. Indicate what shared resources appear in the code and what mechanisms are used to avoid race conditions.

```
void *add (void *argument)
{
  int ct,tmp;
  for (ct=0;ct<REPE;ct++)
  {
     while(test_and_set(&key)== 1);

     tmp = V;
     tmp++;
     V = tmp;
     key = 0;
  }
  printf("->ADD (V=%ld)\n",V);
  pthread_exit(0);
}
```

```
void *subtract (void *argument)
{
  int ct,tmp;
  for (ct=0;ct<REPE;ct++)
  {
     while(test_and_set(&key)== 1);

     tmp = V;
     tmp--;
     V = tmp;
     key = 0;
  }

 printf("->SUBSTRACT (V=%ld)\n", V);
  pthread_exit(0);
}
```

Note. The variables and functions not defined inside functions add and subtract are defined as global

Exercise 6.3:

The following functions are executed by two concurrent threads inside the same process. Indicate what shared resources appear in the code and what mechanisms are used to avoid race conditions.

```
void *add (void *argument)
{
  int ct,tmp;
  for (ct=0;ct<REPE;ct++)
  {
     while(test_and_set(&key)== 1);

     tmp = V;
     tmp++;
     V = tmp;
     key = 0;
  }
  printf("->ADD (V=%ld)\n",V);
  pthread_exit(0);
}
```

```
void *subtract (void *argument)
{
  int ct,tmp;
  for (ct=0;ct<REPE;ct++)
  {
     while(test_and_set(&key)== 1);

     tmp = V;
     tmp--;
     V = tmp;
     key = 0;
  }

 printf("->SUBSTRACT (V=%ld)\n", V);
  pthread_exit(0);
}
```

V : shared Variable: read/write operations from both threads

Protection mechanism: test_and_set function => atomic operation bit test and set processor instruction

- Considers the execution of the following code:

```
#include <stdio.h>
int main()
{
  if (fork() == 0) {
     sleep(10);
  }
  else {
     sleep(5);
  }
  return 0;
}
```

	It generates a zombie child
	It generates an orphan child
	It generates a parent zombie
	It generates a parent orphan
	It doesn't generate any new process

Specify how many processes will be generated as a result of the execution of the following program and how many of them will end up.

```
#include <unistd.h>
int main(void) {
  int i;
  for (i=0; i < 5; i++)
    if (!fork()) break;
  while ( wait(NULL) != -1 );
}
```

Specify how many processes will be generated as a result of the execution of the following program and how many of them will end up.

```
#include <unistd.h>
int main(void) {
  int i;
  for (i=0; i < 5; i++)
    if (!fork()) break;
  while ( wait(NULL) != -1 );
}
```

This program executes 5 times the fork() system call considering that each child abandon the for loop when break is executed. So, 1 + 5 = 6 processes will be executed. All of them will end. Childs directly, and parent when all childs finish (wait).

Given the following C and POSIX code, indicate what will be printed on the standard output by every generated processes. Consider all the possible cases of execution of fork().

```
#include <stdio.h>
#include <unistd.h>
main(){
int  pid;
int i, status;

 i=0;
 printf("Message 1: value %d\n",i);

 switch(pid=fork()){
  case -1: i++;
          printf("Message 2: value %d\n",i);
          exit(-1);
  case 0:  i++;
          printf("Message 3: value %d \n",i);
          exit(3);
  default: i++;
          printf("Message 4: value %d \n",i);
          while (wait(&status)>0);
 }
 printf("Message 5: value %d \n",i);
}
```

Given the following C and POSIX code, indicate what will be printed on the standard output by every generated processes. Consider all the possible cases of execution of fork().

```
#include <stdio.h>
#include <unistd.h>
main(){
int  pid;
int i, status;

 i=0;
 printf("Message 1: value %d\n",i);

 switch(pid=fork()){
   case -1: i++;
            printf("Message 2: value %d\n",i);
            exit(-1);
   case 0:  i++;
            printf("Message 3: value %d \n",i);
            exit(3);
   default: i++;
            printf("Message 4: value %d \n",i);
            while (wait(&status)>0);
 }
 printf("Message 5: value %d \n",i);
}
```

Message 1: value 0
Message 4: value 1 the order of Message 4 and 3 can change
Message 3: value 1
Message 5: value 1

How many processes will be generated by the execution of this program (including the original), as well as which of them will be the last ending? Suppose that none of the used system calls causes errors and that the generated processes do not receive any signal.

```
#include <stdlib.h>
int main(void) {
  int i, pid, pid2;

  for (i=0; i<4; i++) {
    pid=fork();
    if(pid==0) {
      pid2=fork();
      if (pid2!=0) wait(NULL);
      break;
    }
    wait(NULL);
  }
  return 0;
}
```

How many processes will be generated by the execution of this program (including the original), as well as which of them will be the last ending? Suppose that none of the used system calls causes errors and that the generated processes do not receive any signal.

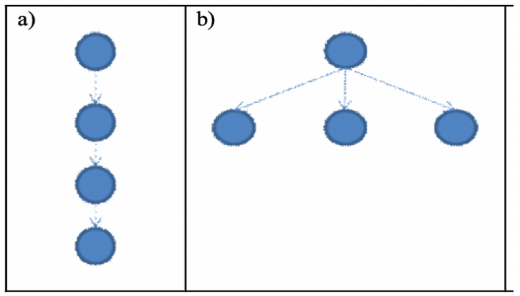
```
#include <stdlib.h>
int main(void) {
  int i, pid, pid2;

  for (i=0; i<4; i++) {
    pid=fork();
    if(pid==0) {
      pid2=fork();
      if (pid2!=0) wait(NULL);
      break;
    }
    wait(NULL);
  }
  return 0;
}
```

8 processes are generated P1..8 + 1 original P0. P0 creates P1 p2 p3 and P4 loop for and pid = fork().
P1 creates P5 (pid2 =fork()) and P1 waits for termination of P5 (that directly executes break and finish) and break to finish. Same for P2 and P6, p3 and P7 and P4 and P8.

Given the following code

```
int main (int argc, char **argv){
  int i;
  int pid;
  for (i=0; i<3; i++){
    pid = fork();
    if (/* CONDITION */) break;
  }
  while( wait(NULL)!=-1);
}
```

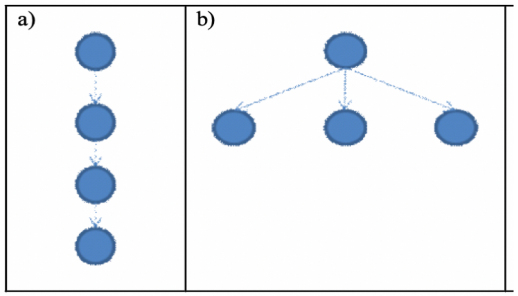
Specify which condition must be met to create a process group with the relationship reflected in each of the following schemes:



Given the following code

```
int main (int argc, char **argv){
  int i;
  int pid;
  for (i=0; i<3; i++){
    pid = fork();
    if (/* CONDITION */) break;
  }
  while( wait(NULL)!=-1);
}
```

Specify which condition must be met to create a process group with the relationship reflected in each of the following schemes:



If (pid != 0) //parent break;	If (pid == 0) // child break

Given the following code

```
#include "all necessary header .h"
int main()
{
  int status;
  printf("Message 1: before exec()\n");
  if (execl("/bin/ps","ps","-la", NULL)<0)
     { printf("Message 2: after  exec()\n");
       exit(1);}

  printf("Message 3: before  exit()\n);
  exit(0);
}
```

Answer the following items:

a) How many processes can be created when it is executed, in what lines are them created and what message prints everyone?

b) When will appear on the standard output the message ”Message 3: before exit()”.

Given the following code

```
#include "all necessary header .h"
int main()
{
  int status;
  printf("Message 1: before exec()\n");
  if (execl("/bin/ps","ps","-la", NULL)<0)
     { printf("Message 2: after  exec()\n");
       exit(1);}

  printf("Message 3: before  exit()\n);
  exit(0);
}
```

Answer the following items:

a) How many processes can be created when it is executed, in what lines are them created and what message prints everyone?

No processes will be created. Exec does not create any process.

The process will print Message 1. If exec fails, then also it will print Message 2

b) When will appear on the standard output the message ”Message 3: before exit()”.

Message 3 will never printed.

Given the following code

```
#include  “all necessary header .h"
main()
{ int i=0;
  int pid;

  while (i<2)
  { switch((pid=fork()))
    {
      case (-1): {printf("Error creating child\n");break;}
      case (0): {printf("Child %i created\n",i);break;}
      default: {printf("Father\n");}
     }
    i++;
  }
  exit(0);
}
```

Answer to:

a) The number of processes generated by its execution and the relationship between them.

b) What messages will appear on the screen if fork() call always succeeds.

.

Given the following code

```
#include  “all necessary header .h"
main()
{ int i=0;
  int pid;

  while (i<2)
  { switch((pid=fork()))
    {
      case (-1): {printf("Error creating child\n");break;}
      case (0): {printf("Child %i created\n",i);break;}
      default: {printf("Father\n");}
     }
    i++;
  }
  exit(0);
}
```

Answer to:

a) The number of processes generated by its execution and the relationship between them.

b) What messages will appear on the screen if fork() call always succeeds.

.

Given the following code

```
#include " all necessary header .h"
#define N 3
main() {
  int i = 0;
  pid_t pid_a;

  while (i<N)
  { pid_a = fork();
    switch (pid_a)
    { case -1:
        printf("Error creating child...\n");
        break;
      case 0:
        printf("Message 1: i = %d \n", i);
        if (i < N-1) break;
        else exit(0);
      default:
        printf("Message 2: i = %d \n", i);
        while (wait(NULL)!=-1);
     }
    i++;
  }
  printf("Message 3: i=%d\n",i);
  exit(0);
}
```

Answer to:

a) Draw the process tree generated when it is running and indicate for every process at what value of "i" it has been created.

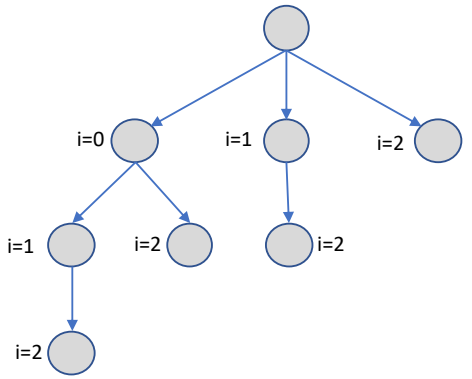
b) Explain if there is a possibility of appearing orphan and/or zombie children.

Given the following code

```
#include " all necessary header .h"
#define N 3
main() {
  int i = 0;
  pid_t pid_a;

  while (i<N)
  { pid_a = fork();
    switch (pid_a)
    { case -1:
        printf("Error creating child...\n");
        break;
      case 0:
        printf("Message 1: i = %d \n", i);
        if (i < N-1) break;
        else exit(0);
      default:
        printf("Message 2: i = %d \n", i);
        while (wait(NULL)!=-1);
     }
    i++;
  }
  printf("Message 3: i=%d\n",i);
  exit(0);
}
```

Answer to:

a) Draw the process tree generated when it is running and indicate for every process at what value of "i" it has been created.



b) Explain if there is a possibility of appearing orphan and/or zombie children.

There is not possibility of orphan child due to each time a fork is invoked the parent wait for the child termination with the sentence:

while (wait(NULL) != -1));

Therefore waiting for the completion of all children guarantees that there are no orphaned, and doing the wait() call immediately after creating child processes ensures that there are no zombies.

Exercise 2

- In a system **without** kernel threads support four threads, H1, H2, H3 and H4 arrive with the following processing demands:

Process	Thread	Arrival	Burst sequence
A	H1	0 (1st)	6 CPU + 2 I/O + 1 CPU
A	H2	0 (2nd)	6 CPU + 2 I/O + 1 CPU
B	H3	0 (3rd)	2 CPU + 3 I/O + 1 CPU + 3 I/O + 1 CPU
B	H4	0 (4th)	2 CPU + 3 I/O + 1 CPU + 3 I/O + 1 CPU

The programming language **run-time** schedules threads applying a **FCFS policy**. I/O is served by a single device with FCFS policy. What will be the **mean waiting time** if the system scheduler uses the following scheduling algorithms?

a) SRTF

b) RR (q=2)

Exercise 2

- In a system **without** kernel threads support four threads, H1, H2, H3 and H4 arrive with the following processing demands:

Process	Thread	Arrival	Burst sequence
A	H1	0 (1st)	6 CPU + 2 I/O + 1 CPU
A	H2	0 (2nd)	6 CPU + 2 I/O + 1 CPU
B	H3	0 (3rd)	2 CPU + 3 I/O + 1 CPU + 3 I/O + 1 CPU
B	H4	0 (4th)	2 CPU + 3 I/O + 1 CPU + 3 I/O + 1 CPU

The programming language **run-time** schedules threads applying a **FCFS policy**. I/O is served by a single device with FCFS policy. What will be the **mean waiting time** if the system scheduler uses the following scheduling algorithms?

a) SRTF

b) RR (q=2)

Exercises: Scheduling threads

SRTF

Process	Thread	Arrival	Burst sequence
A	H1	0 (1st)	6 CPU + 2 I/O + 1 CPU
A	H2	0 (2nd)	6 CPU + 2 I/O + 1 CPU
B	H3	0 (3rd)	2 CPU + 3 I/O + 1 CPU + 3 I/O + 1 CPU
B	H4	0 (4th)	2 CPU + 3 I/O + 1 CPU + 3 I/O + 1 CPU

t	Ready	CPU	I/O queue	I/O	Comment
0					
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					
16					
17					
18					
19					
20					
21					
22					
23					

Exercises: Scheduling threads

Process	Thread	Arrival	Burst sequence
A	H1	0 (1st)	6 CPU + 2 I/O + 1 CPU
A	H2	0 (2nd)	6 CPU + 2 I/O + 1 CPU
B	H3	0 (3rd)	2 CPU + 3 I/O + 1 CPU + 3 I/O + 1 CPU
B	H4	0 (4th)	2 CPU + 3 I/O + 1 CPU + 3 I/O + 1 CPU

SRTF

t	Ready	CPU	I/O queue	I/O	Comment
0	B,A	B(H3)			
1	A	B(H3)			
2	A	A(H1)		B(H3)	
3		A(H1)		B(H3)	
4		A(H1)		B(H3)	
5	A(H1) (BH4)	B(H4)			
6	A(H1)	B(H4)			
7	A(H1)	A(H1)		B(H4)	
8		A(H1)		B(H4)	
9		A(H1)		B(H4)	
10	B(H3)	B(H3)		A(H1)	
11		--	B(H3)	A(H1)	
12	A(H2)	A(H2)		B(H3)	
13		A(H2)		B(H3)	
14		A(H2)		B(H3)	
15	A(H2) B(H4)	B(H4)			
16	A(H2)	A(H2)		B(H4)	
17		A(H2)		B(H4)	
18		A(H2)		B(H4)	
19	B(H3)	B(H3)		A(H2)	
20	B(H4)	B(H4)		A(H2)	
21	A(H1)	A(H1)			FIN DE B
22	A(H2)	A(H2)			
23					FIN DE A

Exercises: Scheduling threads

RR q = 2

Process	Thread	Arrival	Burst sequence
A	H1	0 (1st)	6 CPU + 2 I/O + 1 CPU
A	H2	0 (2nd)	6 CPU + 2 I/O + 1 CPU
B	H3	0 (3rd)	2 CPU + 3 I/O + 1 CPU + 3 I/O + 1 CPU
B	H4	0 (4th)	2 CPU + 3 I/O + 1 CPU + 3 I/O + 1 CPU

t	Ready	CPU	I/O queue	I/O	Comment
0					
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					
16					
17					
18					
19					
20					
21					
22					
23					
24					
25					

Exercises: Scheduling threads

RR q = 2

Process	Thread	Arrival	Burst sequence
A	H1	0 (1st)	6 CPU + 2 I/O + 1 CPU
A	H2	0 (2nd)	6 CPU + 2 I/O + 1 CPU
B	H3	0 (3rd)	2 CPU + 3 I/O + 1 CPU + 3 I/O + 1 CPU
B	H4	0 (4th)	2 CPU + 3 I/O + 1 CPU + 3 I/O + 1 CPU

t	Ready	CPU	I/O queue	I/O	Comment
0	A, B	A (H1)			A, B llegan
1	B	A(H1)			
2	A	B(H3)			
3	A	B(H3)			
4	A	A(H1)		B(H3)	
5		A(H1)		B(H3)	
6	A	A(H1)		B(H3)	
7	B	A(H1)			
8		B(H4)		A(H1)	
9		B(H4)		A(H1)	
10	A	A(H2)		B(H4)	
11		A(H2)		B(H4)	
12		A(H2)		B(H4)	
13	B(H3)	A(H2)			
14	A(H2)	B(H3)			
15		A(H2)		B(H3)	
16		A(H2)		B(H3)	
17		--	A(H2)	B(H3)	
18		B(H4)		A(H2)	
19		--	B(H4)	A(H2)	
20		A(H1)		B(H4)	
21		A(H2)		B(H4)	
22		--		B(H4)	FIN de A
23		B(H3)			
24		B(H4)			
25					FIN de B