Fundamentos de los Sistemas Operativos (FSO)

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

Part 2: Process management

Seminar 5

POSIX threads programming

fSO

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

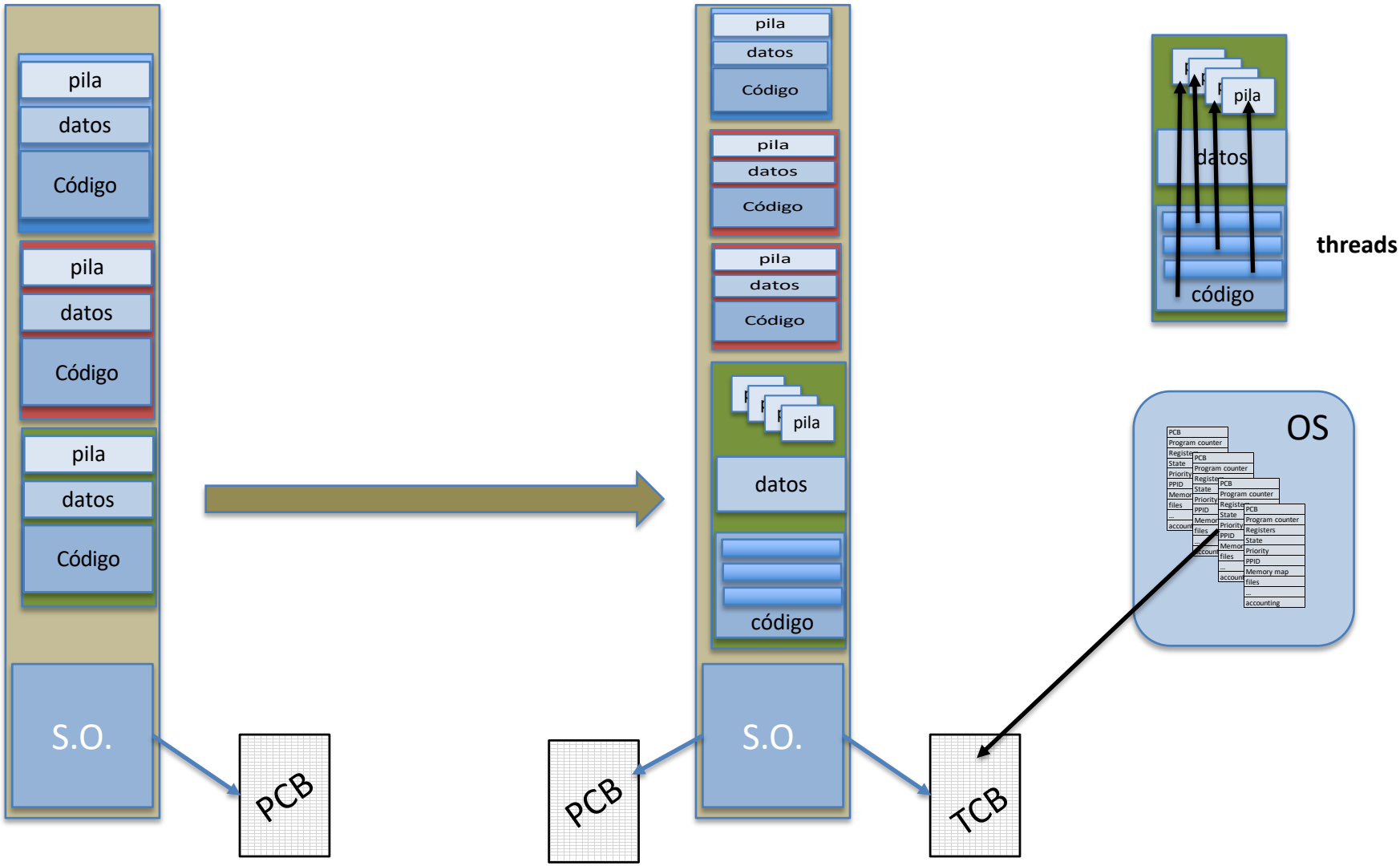
DISCA

- **Goals**
  - Using **POSIX calls** related to **threads creation and basic management**
  - Experimenting **race condition** and the trouble it introduces in concurrent (multithread) programming

- **Bilbiogaphy**
  - "UNIX System Programming", Kay A. Robbins, Steven Robbins. Prentice Hall. ISBN 968-880-959-4 . Chapter 12

- **Introduction**
- Creation
- Ending and waiting
- Identification
- Race condition

Execution thread concept



- Producer and consumer code sketch

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

#define NPRODUCERS 2
#define NCONSUMERS 1
#define MAX_BUFFER 5
#define MAX_ITEMS 20

int  buffer[MAX_BUFFER];
int  input=0, output=0, counter=0;
int  terminados = 0;

int main (int argc, char *argv[])
{

  int t;
  pthread_t threads[NPRODUCERS + NCONSUMERS];

  for(t=0; t<NPRODUCERS; t++){
    pthread_create(&threads[t], NULL, Producer, NULL );
  }

  for(t=0; t<NCONSUMERS; t++){
    pthread_create(&threads[t+NPRODUCERS], NULL, Consumer, NULL
);
  }

  while(terminados < NPRODUCERS + NCONSUMERS) {
   sleep(1);
  }

}
```

```
void *Producer(void *p) {

 int item = 0;
 printf("Producer start\n");
 while(item < MAX_ITEMS) {
  item = item + 1; //produce();
  sleep(1);

  while (counter == MAX_BUFFER)
   /* empty loop */ ;

  buffer[input] = item;
  input = (input + 1) % MAX_BUFFER;
  counter = counter + 1;
  printf("Produced %d size: %d\n", item,
counter);
 }
 printf("Producer finish\n");
 terminados++;
 return 0;
}
```

```
void *Consumer(void *p) {

 int item;
 int nitems = 0;

 printf("Consumer start\n");
 while(nitems < MAX_ITEMS) {
  while (counter == 0)
   /* empty loop */ ;

  item = buffer[output];
  output = (output + 1) % MAX_BUFFER;
  counter = counter - 1;
  nitems++;
  printf("Comsumed %d size: %d\n", item,
counter);
  sleep(2); //consume(item);
 }
 printf("Consumer finish\n");
 terminados++;
 return 0;
}
```

"`counter`" and "`buffer`" are shared by producer and consumer threads
With several producer and consumers, "`input`" is shared by all producers and "`output`" by all consumers

- **POSIX process**
  - It creates an initial thread that executes **main()** function
    - Every thread can create other threads to perform other functions inside the process address space
  - **All threads** inside a process are at the **same level**
    - They are "**brothers**" instead of the "parent-children" relationship in processes
  - All **threads inside a process share global process variables and resources** (files, signal handlers, etc.)
    - Furthermore every thread has a **private copy of its own parameters and local varibles** related to the function it executes

- Basic thread management functions
  - `pthread` library **(`#include <pthread.h>`)**

Function name	Operation
pthread_create	Creates a thread that executes an specified function
pthread_attr_init	Intitializes a thread attribute object to its default values
pthread_attr_destroy	Frees a thread attribute object
pthread_join	Waits for the specified thread to end
pthread_exit	Ends the calling thread
pthread_self	Returns the calling thread ID
pthread_attr_setdetachstate	Changes the detached state attribute
pthread_attr_getdetachstate	Checks the detached state attribute

You have to compile with `-lpthread` option

- Introduction
- **Creation**
- Ending and waiting
- Identification
- Race condition

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void*),
                   void *arg);
```

- `pthread_create ()`
  - It creates a **new thread in ready state**
  - The creator and created threads **compite for the CPU** according to the system scheduling policy (runtime and/or OS)
  - Any thread can call it, not only the main thread
- Arguments
  - `attr`: attribute that features the new thread
  - `start_routine`: function that contains the thread code
  - `arg`: pointer to thread parameters
  - `thread`: is an output argument that is the new thread ID

Creation

- Attributes for thread creation:

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

where

  - `attr` is the attribute to be created/destroyed

- These functions do:
  - Create/destroy a thread creation attribute
  - **`attr_init` initializes `attr` to default values**
    - **Specific functions allow changing attributes values later**
  - The same attribute variable can be reused to create several threads

`pthread_attr_t`

```
Thread attributes:
        Detach state        = PTHREAD_CREATE_JOINABLE
        Scope               = PTHREAD_SCOPE_SYSTEM
        Inherit scheduler   = PTHREAD_INHERIT_SCHED
        Scheduling policy   = SCHED_OTHER
        Scheduling priority = 0
        Guard size          = 4096 bytes
        Stack address       = 0x40196000
        Stack size          = 0x201000 bytes
```

- Changing/checking thread attributes:

```
int pthread_attr_setdetachstate(pthread_attr_t *attr,
                                int detachstate);
int pthread_attr_getdetachstate(const pthread_attr_t *attr,
                                int *detachstate);
```

- where
  - `detachstate` indicates if another thread will be able to wait for the actual thread ending using `pthread_join`
  - Its possible values are:
    - PTHREAD_CREATE_JOINABLE
    - PTHREAD_CREATE_DETACHED

- Example: Hello World

```
//file: pthread_hello.c
//compile: gcc pthread_hello.c -o pthread_hello -lpthread
//see threads in execution: ps -lT
```

```c
#include <stdio.h>
#include <string.h>
#include <pthread.h>

void *My_Print(void *ptr ) {
   char *message;

   message = (char *) ptr;
   write(1,message,strlen(message));
}

int main() {
   pthread_t thread1, thread2;
   pthread_attr_t attr;

   pthread_attr_init(&attr);
   pthread_create(&thread1, &attr, My_Print, "Hello ");
   pthread_create(&thread2, &attr, My_Print, " World\n");

   return 0;
}
```

- Example: 2 periodic Threads

```
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
```

```
struct th_args{
   int tid;
   int sec;
   int ntimes;
} ;

void *PeriodicThread(void *arguments ) {
  int n = 0;
  struct th_args *args = arguments;

  while (n < args->ntimes) {
     printf("[%d]Periodic Thread: %d\n",args->tid, n);
     sleep(args->sec);
     n++;
  }
}

int main() {
  pthread_t thread1, thread2;
  pthread_attr_t attr;
  struct th_args t1, t2;

  pthread_attr_init(&attr);
  t1.tid = 1; t1.sec = 2; t1.ntimes = 10;
  pthread_create(&thread1, &attr, &PeriodicThread, &t1);
  t2.tid = 2; t2.sec = 3; t2.ntimes = 5;
  pthread_create(&thread2, &attr, &PeriodicThread, &t2);
 return 0;
}
```

- Introduction
- Creation
- **Ending and waiting**
- Identification
- Race condition

- **Ending POSIX threads**
  - A thread ends execution by its own when:
    - The thread function ends
    - The thread calls `pthread_exit`

```
int pthread_exit(void *exit_status);
```

where

- `exit_status` is a pointer to a variable by means of which a thread that ends calling `pthread_exit` communicates a ending condition value to another thread waiting to it with `pthread_join`

  - A process ends when its last thread ends

Ending and waiting

- A thread can wait for another to end
  - If the waited thread has been created with the attribute `PTHREAD_CREATE_JOINABLE`
  - To waiting thread should call `pthread_join`

```
int pthread_join(pthread_t thread, void **exit_status);
```

  - `exit_status` is the waited thread returning value through `pthread_exit`

Ending and waiting

- Example

```
...
void *function(void *p) {
   printf("I am a happy brother!\n");
   sleep(10);
}
...
int main( void ) {
   pthread_t       id_thread;
   pthread_attr_t  attributes;

   printf("Main thread: start\n");
   pthread_attr_init(&attributes);
   pthread_create(&id_thread, &attributes, function, NULL);
   printf("Main thread: I have create a brother\n");
   pthread_join(id_thread, NULL);
   printf("Main thread: That´s all folks!");
}
```

¡Warning! We have to declare a variable of thread type for every thread to be created

¿What would be the execution result if we remove pthread_join?

- Introduction
- Creation
- Ending and waiting
- **Identification**
- Race condition

Identification

- Thread identification:

```
pthread_t pthread_self(void);
int pthread_equal(pthread_t th1, pthread_t th2);
```

where

- **`pthread_self`** returns the own thread ID of the calling thread
- **`pthread_equal`** compares two thread ID
  - Thread IDs implementation is unknown
  - It returns **cero** (0) if they are NOT equal and another value if they are equal

Identification

Example: Periodic thread creation

```c
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

void *func_period (void *arg) {
    int period, i;
    period= *((int *)arg);
    for (i=0; i<10; i++) {
        printf("Pthread(period %d):", period);
        printf(" %ld\n", (long) pthread_self());
        sleep (period);
    }
}
```

```c
int main () {
    pthread_t t1,t2;
    pthread_attr_t attr;
    int period1=1, period2=2;

    if (pthread_attr_init(&attr) != 0) {
        printf("Error: atributtes\n");
        exit(1);
    }
    if (pthread_create(&t1, &attr, func_period, &period1) != 0) {
        printf("Error: creating first pthread\n");
        exit(1);
    }
    if (pthread_create(&t2, &attr, func_period, &period2) != 0) {
        printf("Error: creating second pthread\n");
        exit(1);
    }
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
}
```

```
//file: th_periodic.c
//compile: gcc th_periodic.c -o th_periodic -lpthread
//see threads in execution: ps -lT
```

- Introduction
- Threads creation
- Threads ending
- Waiting
- Thread identification
- **Race condition**

Race condition

Example "globalvar.c" sequential

- It increments 40.000.000 times a global variable
  - Then the final result must be 40.000.000

```
//file: globalvar.c
//compile: gcc globalvar.c –o globalvar
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int Globalvariable;

int main() {
    int i;
    long iterations = 40000000;
    for (i=0; i<(iterations); i++) {
        variableGlobal ++;
    }
    printf("Globalvariable= %d\n",Globalvariable);

    return 0;
}
```

Race condition

- **Example** race_condition.c
  - "globalvar.c" concurrent
  - Two threads cooperate in incrementing the global variable
  - Every thread does 20.000.000 operations
  - At the end:

    Gobalvariable = 40000000

```
//file: race condition.c
//compile: gcc race_condition.c -o race_condition -lpthread
```

```
#include <stdio.h>
#include <pthread.h>

int Globalvariable;

void *Addition(void *ptr) {
    int i, aux_variable;
    int *iter = (int *)ptr;
    for (i=0; i<*iter; i++){
        aux_variable = Globalvariable;
        aux_variable++;
        Globalvariable = aux_variable;
    }
}
```

```
int main() {
    long iterations = 20000000;
    pthread_t t1, t2;
    pthread_attr_t attr;

    pthread_attr_init(&attr);
    pthread_create(&t1, &attr, Addition, &iterations);
    pthread_create(&t2, &attr, Addition, &iterations);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Globalvariable= %d\n", Globalvariable);
    return 0;
}
```

Is ALWAYS execution result 40.000.000??

- Given the following C and POSIX code

```
1  /***Example4.c ***/
2  #include " all necessary header .h"
3  #define N 3
4  main() {
5    int i = 0;
6    pid_t pid_a;
7
8    while (i<N)
9    { pid_a = fork();
10     switch (pid_a)
11     { case -1:
12         printf("Error creating child...\n");
13         break;
14       case 0:
15         printf("Message 1: i = %d \n", i);
16         if (i < N-1) break;
17         else exit(0);
18       default:
19         printf("Message 2: i = %d \n", i);
20         while (wait(NULL)!=-1);
21      }
22     i++;
23   }
24   printf("Message 3: i=%d\n",i);
25   exit(0);
26 }
```

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

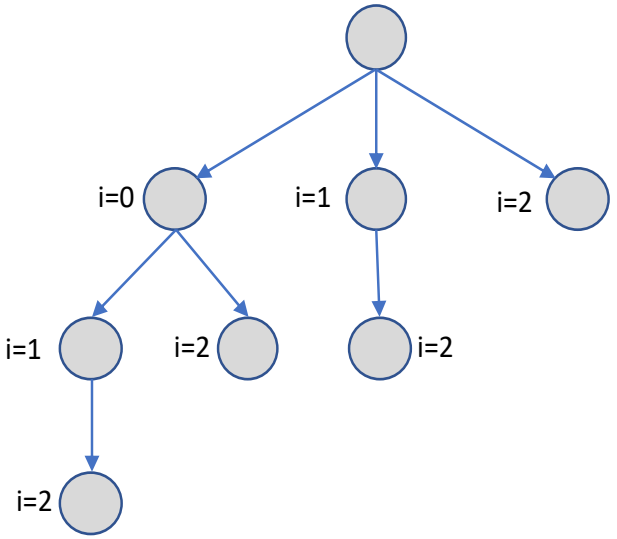
b) Explain if there is a possibility of appearing orphan and/or zombie children.

a) Indicate what changes are required in the previous code in order to make the child process orphan and to be adopted by `init`. (**Note**: Use sleep() and C sentences).
To generate an orphan child and be inherited by init process, the parent process has to exit before the child.
The code to be added in line 7 is
If (val != 0) exit(0);

b) Indicate what changes are required in the previous code in order to make the child process zombie for a while. (**Note**: Use sleep() and C sentences).
To generate a zombie child it is needed that child exit when parent is not waiting for the child termination.
The code to be added in line 7 is
If (val == 0) {exit(0);
Insert line 10 (before return 0) wait(NULL);

- Given the following C and POSIX code

```
1  /***Example4.c ***/
2  #include " all necessary header .h"
3  #define N 3
4  main() {
5    int i = 0;
6    pid_t pid_a;
7
8    while (i<N)
9    { pid_a = fork();
10     switch (pid_a)
11     { case -1:
12         printf("Error creating child...\n");
13         break;
14       case 0:
15         printf("Message 1: i = %d \n", i);
16         if (i < N-1) break;
17         else exit(0);
18       default:
19         printf("Message 2: i = %d \n", i);
20         while (wait(NULL)!=-1);
21      }
22     i++;
23   }
24   printf("Message 3: i=%d\n",i);
25   exit(0);
26 }
```



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

- Given the following C and POSIX code

```
1   /***Example5.c ***/
2   #include " all necessary header .h"
3
4   int main()
5   { pid_t val;
6      printf("Message 1: before exec()\n");
7      fork();
8      execl("/bin/ls","ls","-la", NULL);
9      val = fork();
10     if (val==0)
11     {execl("/bin/ps","ps","-la", NULL);
12      printf("Message 2: after  exec()\n");
13         exit(1)
14      }
15     printf("Message 3: before exit()\n);
16    exit(0);
    }
```

a) Explain the number of processes that are creating when executing Test and the parent/child relationship.
b) Indicate and explain the messages and information shown on the screen when executing Test.

a) Explain the number of processes that are creating when executing Test and the parent/child relationship.
Only 1 process is created (fork() line 7. In line 8 both processes (parent and child) change its image to exec and execute ls -la. None of the 2 processes can execute lines from 8.
b) Indicate and explain the messages and information shown on the screen when executing Test.
In the console will appear:
`Message 1: before exec()`
And the result of the command ls -al twice.

- Given the following C and POSIX code

```
   /***Example6.c***/
   #include " all necessary header .h"
1
2  int main(void)
3  { int val;
4    printf("Message 1\n");
5    val=fork();
6    /** Write here your changes **/
7    sleep(5);
8    printf("Message 2\n");
9    return 0;
10 }
11
```

a) Indicate what changes are required in the previous code in order to make the child process orphan and to be adopted by init. (**Note**: Use sleep() and C sentences).

B) Indicate what changes are required in the previous code in order to make the child process zombie for a while. (**Note**: Use sleep() and C sentences).

Exercise 1

- In a system with kernel threads support four threads, H1, H2, H3 and H4 arrive with the following processing demands:

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

I/O is performed in one single device and delivers access following FCFS policy. What will be the **mean waiting time** if the scheduler uses the following algorithms?

a) SRTF

b) RR (q=2)

Exercises: Scheduling threads

Exercise 1

- SRTF (Shortest-remaining-time-first)

Thread	Arrival	Burst sequence
H1	0 (1st)	6 CPU + 2 I/O + 1 CPU
H2	0 (2nd)	6 CPU + 2 I/O + 1 CPU
H3	0 (3rd)	2 CPU + 3 I/O + 1 CPU + 3 I/O + 1 CPU
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	H2(6),H1(6),H4(2)	H3(1)			Arrive H1, H2, H3, H4
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					
16					
17					
18					
19					
20					
21					
22					
23					
24					

H1	7
H2	15
H3	0
H4	2
	24
	6

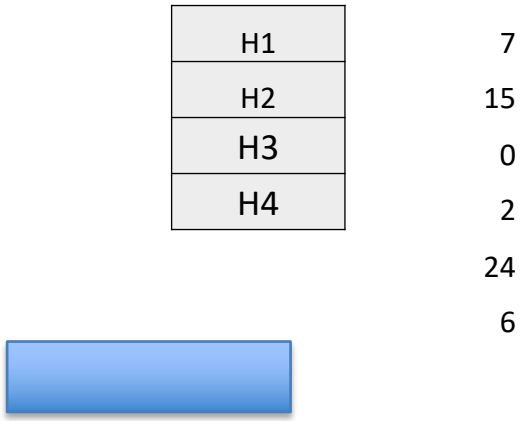
Exercises: Scheduling threads

Exercise 1

- SRTF (Shortest-remaining-time-first)

Thread	Arrival	Burst sequence
H1	0 (1st)	6 CPU + 2 I/O + 1 CPU
H2	0 (2nd)	6 CPU + 2 I/O + 1 CPU
H3	0 (3rd)	2 CPU + 3 I/O + 1 CPU + 3 I/O + 1 CPU
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	H2(6),H1(6),H4(2)	H3(1)			Arrive H1, H2, H3, H4
1	H2(6),H1(6),H4(2)	H3(0)			CPU => H3
2	H2(6),H1(6)	H4(1)		H3(2)	
3	H2(6),H1(6)	H4(0)		H3(1)	CPU => H4
4	H2(6)	H1(5)	H4(3)	H3(0)	IO => H3
5	H2(6), H1(5)	H3(0)		H4(2)	CPU => H3
6	H2(6)	H1(4)	H3(3)	H4(1)	
7	H2(6)	H1(3)	H3(3)	H4(0)	IO=>H4
8	H2(6), H1(3)	H4(0)		H3(2)	CPU => H4
9	H2(6)	H1(2)	H4(3)	H3(1)	
10	H2(6)	H1(1)	H4(3)	H3(0)	IO => H3
11	H2(6),H1(1)	H3(0)		H4(2)	Finish H3
12	H2(6)	H1(0)		H4(1)	CPU => H1
13		H2(5)	H1(2)	H4(0)	
14	H2(5)	H4(0)		H1(1)	Finish H4
15		H2(4)		H1(0)	IO=>H1
16	H2(4)	H1(0)			Finish H1
17		H2(3)			
18		H2(2)			
19		H2(1)			
20		H2(0)			CPU => H2
21		-		H2(1)	
22		-		H2(0)	IO => H2
23		H2(0)			Finis H2
24					



Exercises: Scheduling threads

Exercise 1

- RR q = 2

Thread	Arrival	Burst sequence
H1	0 (1st)	6 CPU + 2 I/O + 1 CPU
H2	0 (2nd)	6 CPU + 2 I/O + 1 CPU
H3	0 (3rd)	2 CPU + 3 I/O + 1 CPU + 3 I/O + 1 CPU
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					

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

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(H3)			
1	A	B(H3)			
2	A	A(H1)		B(H3)	
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

t	Ready	CPU	I/O queue	I/O	Comment
0	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)	Finish B
21	A(H1)	A(H1)			
22	A(H2)	A(H2)			Finish A
23