OPERATING SYSTEMS

Module2_Part6

Textbook : Operating Systems Concepts by Silberschatz

Scheduling algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue

is to be allocated the CPU. There are many different CPU-scheduling algorithms. Some of

them are

First-Come, First-Served Scheduling

Shortest-Job-First Scheduling

shortest-remaining-time-first

Priority Scheduling

Round-Robin Scheduling

Shortest job first scheduling algorithm

The shortest-job-first (SJF) scheduling algorithm.

- associates with each process the length of the process's next CPU burst.

-When the CPU is available, it is assigned to the process that has the smallest next CPU burst.

If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.

Assumption: run time for processes are known in advance

Shortest Job First yields smallest average turnaround time, if all jobs are available simultaneously.

SJF

- SJF is an optimal algorithm because it decreases the wait times for short processes much

  more than it increases the wait times for long processes.

  It gives minimum turn around time

Consider the case of 4 jobs, with run times of a,b,c,and d respectively.

The first job finishes with time a,the second job finishes with time a+b and so on.

The average turn around time =(4a+3b+2c+d)/4. It is clear that 'a' contributes to the average

than the other times, so it should be the shortest job ,with b next, then c and so on. So we

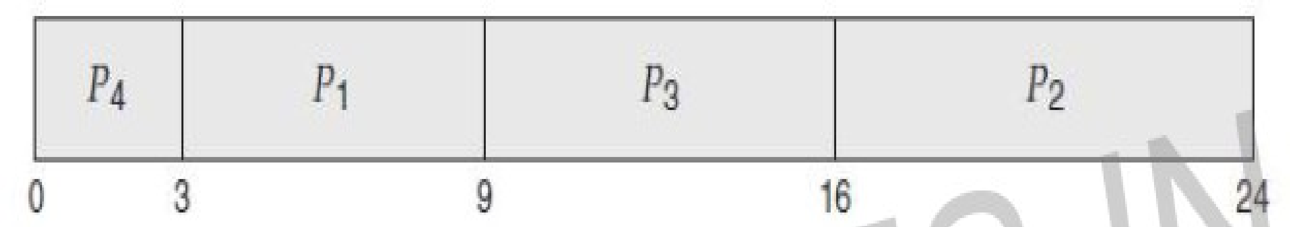
can say that SJF is optimal

SJF

- As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time
*P*1	6
*P*2	8
*P*3	7
*P*4	3

- Using SJF scheduling, we would schedule these processes according to the following Gantt
- chart:

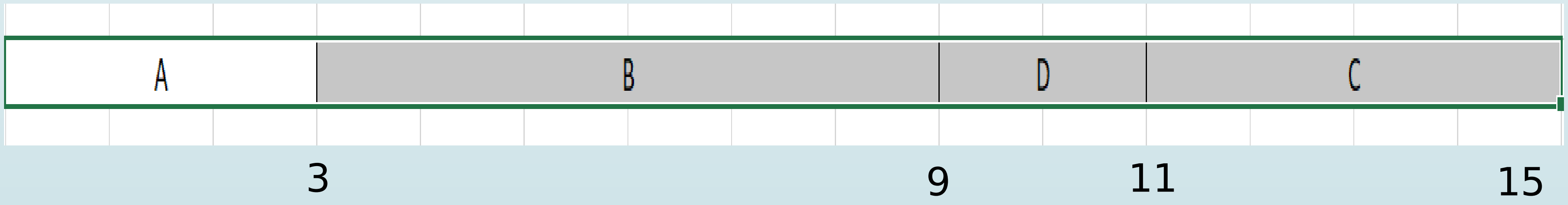


The waiting time is 3 milliseconds for process *P*1, 16 milliseconds for process *P*2, 9
milliseconds for process *P*3, and 0 milliseconds for process *P*4.
Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7
milliseconds. By comparison, if we were using the FCFS scheduling

SJF

For the processes listed draw gantt chart illustrating their execution

process	Arrival time	Processing time
A	0	3
B	1	6
C	4	4
D	6	2

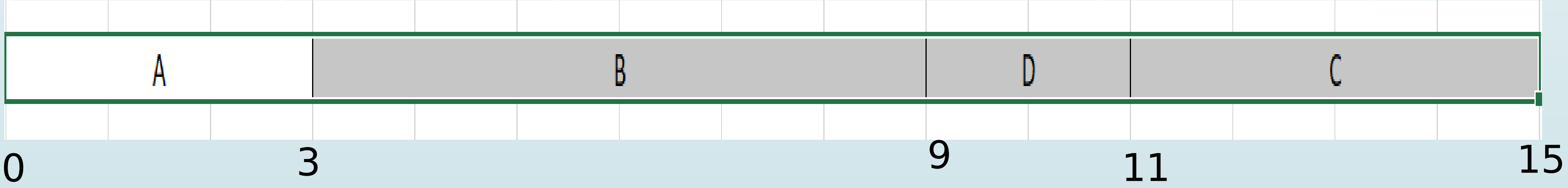


Process A start executing: It is the only choice at time 0 . At time 3, B is the only choice .At time 9, B completes, process D runs because D is shorter than process C

SJF

- For the process listed what is the average turn around time?

Process	Arrival time	Processing time	Completion time	Turn around time
A	0	3	3	3
B	1	6	9	8
C	4	4	15	11



Turn around time=completion time –arrival time

Average turn around time=((3-0)+(9-1)+(15-4)+(11-6))/4 = 6.75

SJF

- For the processes listed what is the waiting time for each process?

Process	Arrival time	Processing time	Completion time	Turn around time	Waiting time
A	0	3	3	3	3
B	1	6	9	8	8
C	4	4	15	11	11
D	6	2	11	5	5

Waiting time=turn around time – execution time

A:(3-3)=0   B:(8-6)=2   C:(11-4)=7   D:(5-2)=3

fork() system call

fork() system call is used to create child processes in a C program.

It takes no arguments and returns a process ID.

After a new child process is created, ***both*** processes will execute the next instruction following the ***fork()*** system call.

Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of **fork().**

If **fork()** returns a negative value, the creation of a child process was unsuccessful.

fork() returns a zero to the newly created child process.

fork() returns a positive value, the ***process ID*** of the child process, to the parent.

fork() system call

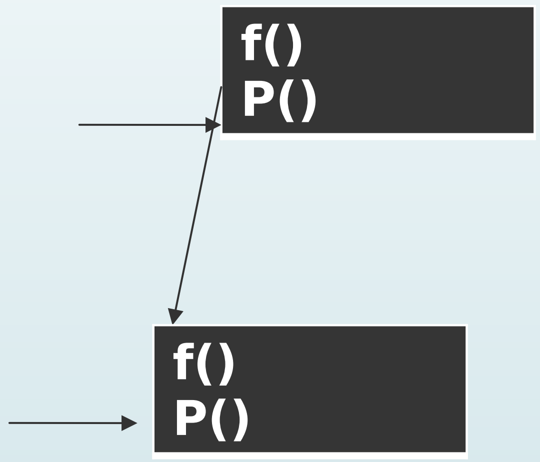
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{

    // make two process which run
same
    // program after this
instruction
    fork();

    printf("Hello world!\n");
    return 0;
}
```

output

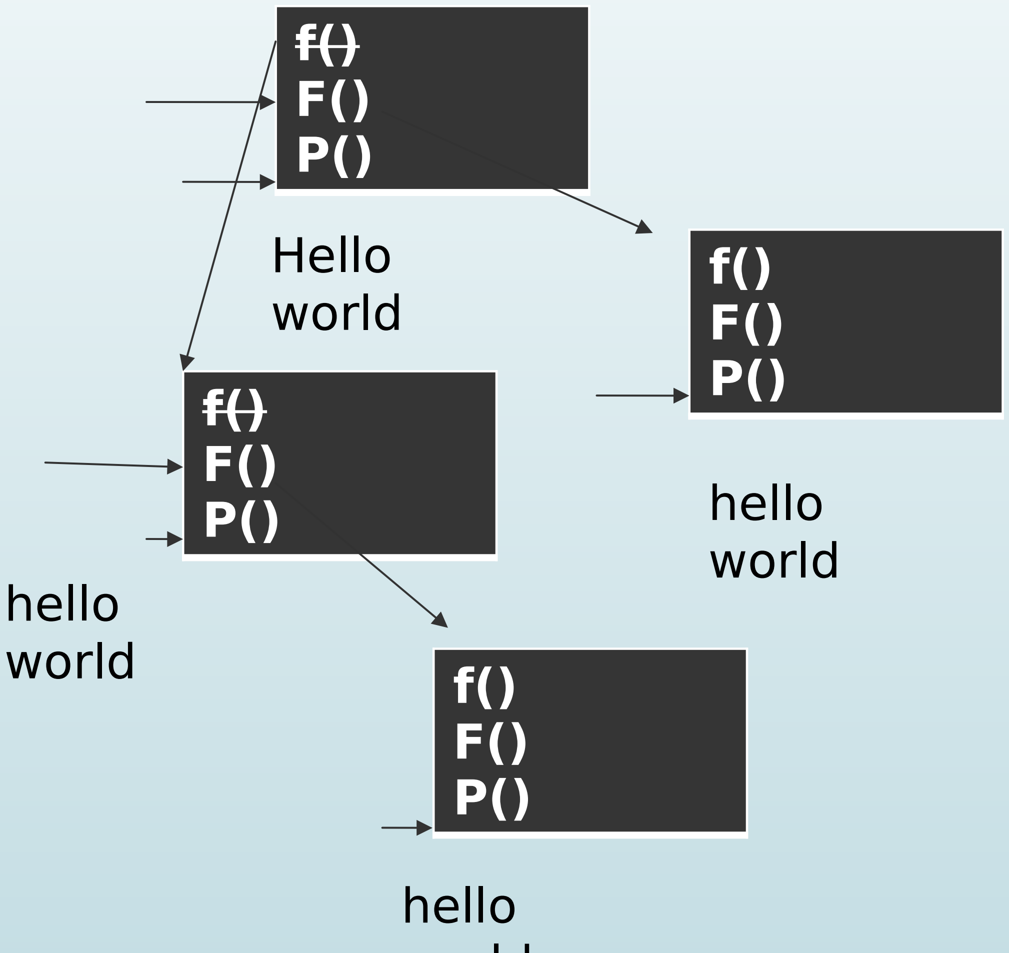
```
Hello world!
Hello world!
```



fork() system call

- Consider this code

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
main()
{
fork();
fork();
printf(“hello world”);
}
```



Four times hello world will be printed

getpid() system call

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int  main(void)
{
//variable to store calling function's process id
 pid_t process_id; // pid_t unsigned integer type
//variable to store parent function's process id
 pid_t p_process_id;


//getpid() - will return process id of calling function
 process_id = getpid();
//getppid() - will return process id of parent function
p_process_id = getppid();

//printing the process ids
printf("The process id: %d\n",process_id);
printf("The process id of parent function: %d\n",p_process_id);
return 0;
}
Output

The process id: 31120
The process id of parent function: 31119
```

Exec system call

Exec system call

The exec() system call is used to execute a file which is residing in an active process. When exec() is called the previous executable file is replaced and new file is executed.

process id will be the same.

Exec() system call

- There are two programs ex1.c ex2.c

Ex1.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
Int main(int argc, char  *argv[])
{
printf(“Pid of ex1.c=%d\n”,getpid());
Char *args[] ={hello”,NULL};
execv(“./ex2”,args);
printf(“Back to Ex1.c”);
Return 0;
}
```

Exec sytem call

- Ex2.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
Int main(int argc,char  *argv[])
{
printf(“We are in ex2.c\n”);
printf(“Pid of ex2.c=%d\n”,getpid());
Return 0;
}
```

- Compile these two programs
- gcc ex1.c -o ex1
- gcc ex2.c –o ex2
- Run the first program ./ex1

- Pid of ex1.c=5962
- We are in ex2.c
- Pid of ex2.c=5962

Wait system call

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio..h>
Int main()
{
        pid_t  q;
        q=fork();
        if(q==0)//child
        {       printf("I am a child having Id  %d/n",getpid());
                 printf("My parent's id is %d\n",getppid());
        }
        else{//parent
                printf(" My child's id is   %d/n",q));
                 printf("I  am parent having  id  %d\n",getpid());
        }
        printf("Common");
}
```

A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent ***continues*** its execution after wait system call instruction.

Wait system call

- Output may be

My child’s id is 188
I am a child having Id 188
I am parent having id 157
My parent’s id is 157
Common
Common

Wait system call

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio..h>
#include <sys/wait.h>
Int main()
{
        pid_t  q;
        q=fork();
        if(q==0)//child
        {           printf("I am a child having Id  %d/n",getpid());
         printf("My parent's id is %d\n",getppid());
        }
        else{//parent
        wait(NULL);
        printf(" My child's id is   %d/n",q));
         printf("I  am parent having  id  %d\n",getpid());
        }
        printf("Common");
}
```

Wait system call

when compiles and run

I am a child having Id 256
My parent’s id is 255
Common
My child’s id is 256
I am parent having id 255
Common

exit()

- It deletes all buffers and closes all open files before ending the program.

```
// C program to illustrate exit() function.
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    printf("START");

    exit(0); // The program is terminated here

    // This line is not printed
    printf("End of program");

}
```

Output
START