Volume 5, Issue 10, October-2015 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Executing a PCB Suite on Raspberry pi Scheduled with Priorities using Process Synchronization

Rahul Jassal
Panjab University Swami Sarvanand Gir Regional Centre,
Hoshiarpur, India

Abstract: ***The paper describes a process scheduling concept that is executed on the basis of time and on the preference of their priority so that it is synchronized with CPU. The Operating System maintains the status of CPU in collaboration with scheduler, which maintains a copy of ready queue of all the processes to be executed and handover to subroutines scheduled with Raspberry pi . If any Interrupt occurs then it will terminate the whole process without its completion. Also the major problem that comes under this scheduling is indefinite blocking or starvation. As first process will gets interrupted so it will not provide the CPU to next process.***

Keywords: Raspberrypi, int piHiPri (int priority), scheduler, PCB, int piThreadCreate (name)

I. INTRODUCTION

Scheduling is to manage the processes in different ways. But in this scheduling, different processes have been assigned different priorities; processes with higher priorities are carried out first whereas processes having equal priorities are carried out on the basis of FCFS (First Come First Serve). The main aim of this scheduling is to maximize the CPU utilization and the throughput such that the turnaround time is linearly proportional to total execution time. There are two scheduling techniques- Preemptive or non-preemptive. Preemptive is that, when a process switch from running state to ready state and non-preemptive is that, when a process switches from running state to ready state. This scheduling comes under both the scheduling techniques because it depends not only on criteria but on two values: priority and burst time. Priority is the rank given to each process and burst time is the time taken by the CPU to complete the whole process.

In first column P_A is passed from the entry section and is in processing state now. In second column P_A is present in the critical section and is in executing state and after completing its task, it will terminate. In the third column P_C is passed in the entry section, at the same time P_B is passed in the critical section which is impossible because it is not possible to pass two processes in the same section at a time. In the forth column also P_C is passed in the entry section and P_B is passed in the critical section and remainder and exit sections of P_B are false but as a process cannot go into the critical section without passing the entry section, it causes the interrupt because is P_C is passed in the entry section but not P_B so P_C should be false in remainder and exit section. In the last column, P_C is false in all the sections from entry to exit. If any Interrupt occurs then it will terminate the whole process without its completion. This is known as starvation.

Process's Status in different Time Intervals…

Priority Wise	Time	Critical Section	Entry Section	Exit Section	Remainder Section	Remarks
	T0	PA(True)	PA(Passed)	PA(False)	PA(False)	PA is in processing
	T0	PA(passed)	PA(Passed)	PA(True)	PB(False)	PA is terminated
	T1	PB(True)	PC(passed)	PC(False)	PC(False)	Not possible
	T1	PB(True)	PC(Passed)	PB(False)	PB (False)	Interrupt
	T1	PC(False)	PC(False)	PC(False)	PC(False)	Starvation

In this types of CPU scheduling, time and priority acts as a constraint also and only those processes that are present in the CPU scheduler can participate in CPU Scheduling process. If any Interrupt occurs then it will terminate the whole process without its completion. Also the major problem that comes under this scheduling is indefinite blocking or starvation. As first process will gets interrupted so it will not provide the CPU to next process.

Priority, Interrupts and Threads

Using **WiringPi** one can make use of functions to manage control of program (or thread) priority and even used for creating new thread inside a program. More than one thread can be run concurrently with main program and used for variety of purposes.

II. GENERICS

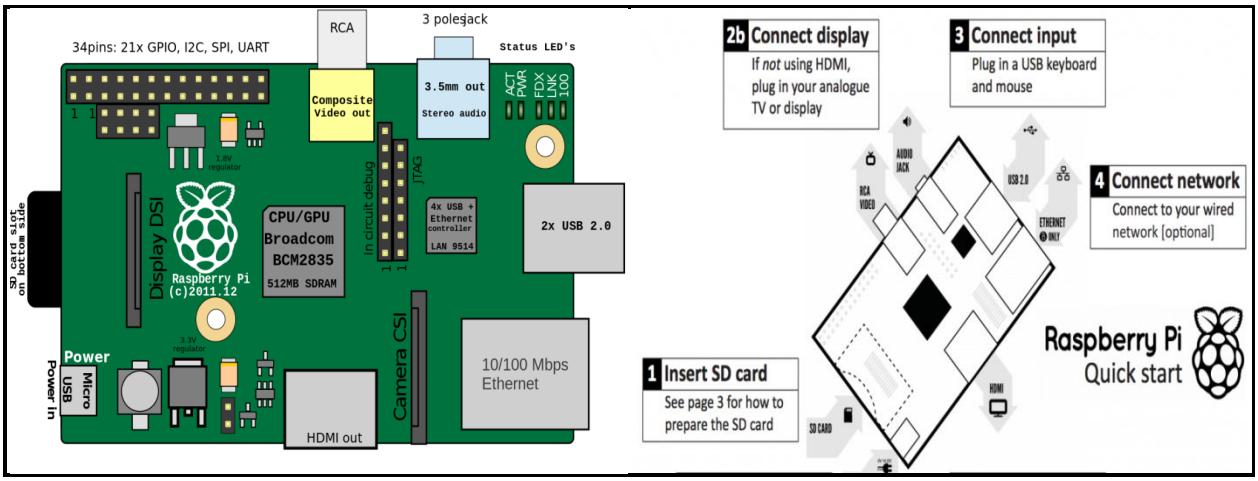
```
{
process_A *p,p1;
```



```
process_B p2;
process_C p3;
int arr[10], bursttime[10],priority[10],waitingtime[10],turnaroundtime[10],process[10],i,j,n;
int total=0,pos,awt,att,temp;
clrscr();
cout<<"\t\t\t     WELCOME TO";//sleep(1);
cout<<"\n"<<"\t\t\tPROCESS SCHEDULING";//sleep(1);
cout<<"\n \t\tWE ARE WORKING IN PRIORITY SCHEDULING";//sleep(1);
cout<<"\n*********************************************************************************"; sleep(3);
cout<<"\n the total no. of processes";
cin>>n;
cout<<"\n enter burst time and priority \n";
for(i=0;i<n;i++)
{
cout<<"\n p["<<i+1<<"] \n";
cout<<"bursttime";
cin>>bursttime[i];
//bursttime[i]=rand()%3;
cout<<"\n priority\n";
cin>>priority[i];
//priority[i]=rand()%3;
arr[i]=priority[i];
process[i]=i+1;
}
for(i=0;i<n;i++)
{
pos=i;
for(j=i+1;j<n;j++)
{
if(priority[j]<priority[pos])
pos=j;
}
temp=priority[i];
priority[i+1]=priority[pos];
priority[pos]=temp;
temp=bursttime[i];
bursttime[i]=bursttime[pos];
bursttime[pos]=temp;
temp=process[i];
process[i]=process[pos];
process[pos]=temp;
}
waitingtime[0]=0;
for(i=1;i<n;i++)
{
waitingtime[i]=0;
for(j=0;j<i;j++)
waitingtime[i]=waitingtime[i]+bursttime[j];
total=total+waitingtime[i];
}
awt=total/n;
total=0;
sleep(1);
cout<<"\n*********************************************************************************";  sleep(1);
cout<<"u had assign priority to each process";// sleep(1);
cout<<"\ncalculation of burst time and turn around time as follows";// sleep(1);
cout<<"\n*********************************************************************************";  sleep(1);
cout<<"\n \tprocess\t ";//sleep(1);
cout<<"\tpriority\t";//sleep(1);
cout<<"\twaitingtime \t";//sleep(1);
cout<<"turnaroundtime";
for(i=0;i<n;i++)
{
```



```
turnaroundtime[i]=bursttime[i]+waitingtime[i];
total=total+turnaroundtime[i];
cout<<"\n \t process["<<process[i]<<"]\t\t";//sleep(1);
cout<<bursttime[i];sleep(1);
cout<<"\t\t"<<waitingtime[i];sleep(1);
cout<<"\t\t"<<turnaroundtime[i];   // sleep(1);
}
att=total/n;
cout<<"\n*********************************************************************************************";//
sleep(3);
cout<<" average waiting time="<<awt;//sleep(1);
cout<<"\n*********************************************************************************************";//
sleep(3);
cout<<"average turn around time="<< att;  //  sleep(1);
cout<<"\n*********************************************************************************************";//
sleep(3);
cout<<"\n\t\t\t\t DONE";
  if(priority[0]>=priority[1] && priority[0]>=priority[2])
{
p=&p1;
p->critical();
p->exit();
}
if(priority[1]>=priority[0] && priority[1]>=priority[2])
{
p=&p2;
p->critical();
p->exit();
}
if(priority[2]>=priority[0] && priority[2]>=priority[1])
{
p=&p3;
p->critical();
p->exit();
}
```



III. PROGRAM OR THREAD PRIORITY

- **int piHiPri (int priority) ;**

The above function helps to switch a program (or thread in a multi-threaded program)[1][2] to a higher priority and enables a real-time scheduling. The **priority** parameter should be from 0 (the default) to 99 (the maximum). It doesn't enhances the performance of program but assigned a bigger slice of time when other programs are run simultaneously. The priority parameter works relative to others – so we can make one program priority 1 and another priority 2 and it will have the same effect as setting one to 10 and the other to 90 (as long as no other programs are running with elevated priorities)

The return value is 0 for success and -1 for error. If an error is returned, the program should then consult the *errno* global variable, as per the usual conventions. Only programs running as root can change their priority. If called from a non-root program then nothing happens.

Interrupts

The newer kernel wrapped with GPIO interrupt handling code, which frees up the processor to do other tasks while you're waiting for that interrupt. The GPIO can be set to interrupt on a rising, falling or both edges of the incoming signal.

- . **int waitForInterrupt (int pin, int timeOut) ;**

During the call of the function, it will wait for an interrupt event to happen on that pin and your program will be stalled. The **timeOut** parameter is given in milliseconds, or can be -1 which means to wait forever.

The return value is -1 if an error occurred (and *errno* will be set appropriately), 0 if it timed out, or 1 on a successful interrupt event. After initializing the GPIO pin, the waitForInterrupt function is called and at present the only way to do this is to use the **gpio** program, either in a script, or using the system() call from inside your program.

gpio edge 0 falling before running the program.

- **int wiringPiISR (int pin, int edgeType, void (*function)(void)) ;**

The above function registers a function to received interrupts on the specified pin. The edgeType parameter is either **INT_EDGE_FALLING**, **INT_EDGE_RISING**,**INT_EDGE_BOTH** or **INT_EDGE_SETUP**. If it is **INT_EDGE_SETUP** then no initialisation of the pin will happen – it's assumed that you have already setup the pin elsewhere (e.g. with the ***gpio*** program), but if you specify one of the other types, then the pin will be exported and initialised as specified. This is accomplished via a suitable call to the ***gpio*** utility program, so it need to be available.

The pin number is supplied in the current mode – native wiringPi, BCM_GPIO, physical or Sys modes. The function will be called when the interrupt triggers. When it is triggered, it's cleared in the dispatcher before calling your function, so if a subsequent interrupt fires before you finish your handler, then it won't be missed. (However it can only track one more interrupt, if more than one interrupt fires while one is being handled then they will be ignored)This function is run at a high priority (if the program is run using sudo, or as root) and executes concurrently with the main program. It has full access to all the global variables, open file handles and so on.

- **int piThreadCreate (*name*) ;**

The function creates a thread which is another function in your program previously declared using the **PI_THREAD** declaration. This function is then run concurrently with your main program. An example may be to have this function wait for an interrupt while your program carries on doing other tasks. The thread can indicate an event, or action by using global variables to communicate back to the main program, or other threads.

IV. CONCLUSION

Scheduling is a fundamental operating system function. A Scheduler should consider fairness, efficiency, response time, turnaround, throughput etc. If a Process wants to take the full control on the processor so that the CPU cannot be taken away from that process, then that time non-preemptive scheduling is used. A scheduling discipline is preemptive if, once a process has been given the CPU can taken away. In PRIORITY SCHEDULING, priority matters but along with that its burst time also matters a lot. After priority and burst time, the next thing that matter is FIFO process. One can make use of multi threading on such controllers and may opt for IoT in future.

REFERENCES

[1] Multitasking/multithreading in C on Arduino, **http://playwithmyled.com/2009/10/multitasking-multithreading-in-c-on-arduino/, 2009**

[2] Multithreaded C Program on the Raspberry PI, **http://stackoverflow.com/questions/18423885/multithreaded-c-program-on-the-raspberry-pi, 2015**

AUTHORS PROFILE

Rahul Jassal is working as Assistant Professor in Department of Computer Science & Application, Panjab University Regional Centre, Hoshiarpur, India. He received his Master Degree in year 2007 and qualifies the lectureship examination for subject "Computer Science & Application in the same year. He area of interest lies with algorithms designs & analysis.

