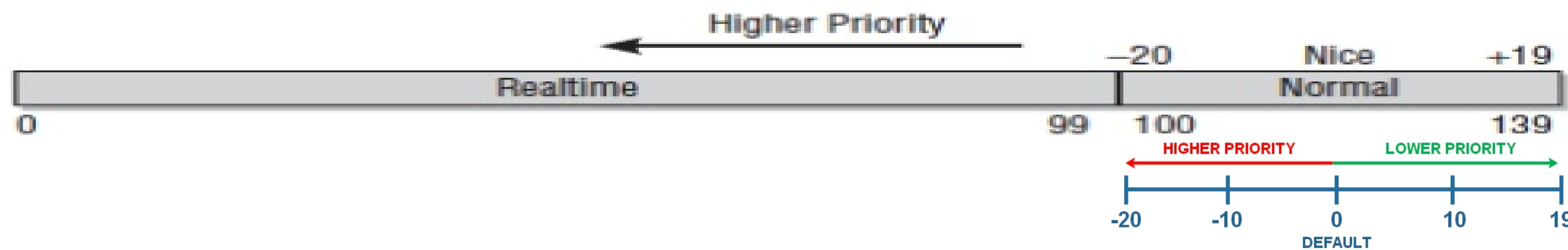
Real Time Systems and Control Applications

Contents

Priority in OS

Linux Priority Levels and Nice Values

- Linux has static priority ranged from 0 to 139, where 0 to 99 are reserved for real time tasks, and 100 to 139 for users.



- The priority is represented as a nice value (niceness) from [-20, 19] mapping to priority level 100 to 139.

- The lower the NICE (or priority) value, the higher priority a process gets. By default the priority Nice value is zero.

Changing Process Priority

- nice --- change process priority on Linux
- Example:

  nice –10 <aProcess>: set the process with a priority which has nice value 10

  nice –n –10 <aProcess>: increase the nice value of a process by 10

  (note that – is a hyphen, not negative sign)

  Wait, does Linux allow you to change a process priority?

  It is fine to set a process with a lower priority, but you will need a superuser's privilege to set a higher priority of a process.

Get and Set Process Priority in C

```
#include <sys/resource.h>

int getpriority(int which, id_t who);

int setpriority(int which, id_t who, int value);
```

The value *which* is one of
PRIO_PROCESS, **PRIO_PGRP**, or **PRIO_USER**

Code Example

```
#include <sys/resource.h>
...
int which = PRIO_PROCESS;
id_t pid;
int ret;
pid = getpid();
int priority = -20;

ret = getpriority(which, pid);
ret = setpriority(which, pid, priority);
...
```

Alternatively call nice() in your program

“int nice(int inc);” increases the process priority by *inc*.

Example:

nice(10);

This sets the priority of the process as current priority+10

unprivileged user can only lower the process priority.

Code Example 1:
fork() without explicitly assigning priority

```
#include  <stdio.h>
#include  <string.h>
#include  <sys/types.h>

void  main(void){
     pid_t  pid;
     int    i=0, num =10;

     pid = fork();
     if (pid == 0){
        pid = getpid();
        for (i = 1; i <= num; i++)
              printf("This line is in Child Process from pid %d, iteration = %d\n", pid, i);
     }
     else{
        pid = getpid();
        for (i = 1; i <= num; i++) {
              printf("This line is in Parent Process from pid %d, iteration = %d\n", pid, i);
        }
     }
}
```

Child Process

Parent Process

Output of the Code

```
[hewll@mills ~/test/process] ./forkTest
This line is in Parent Process from pid 6584, iteration = 1
This line is in Parent Process from pid 6584, iteration = 2
This line is in Parent Process from pid 6584, iteration = 3
This line is in Parent Process from pid 6584, iteration = 4
This line is in Parent Process from pid 6584, iteration = 5
This line is in Parent Process from pid 6584, iteration = 6
This line is in Parent Process from pid 6584, iteration = 7
This line is in Parent Process from pid 6584, iteration = 8
This line is in Parent Process from pid 6584, iteration = 9
This line is in Parent Process from pid 6584, iteration = 10
This line is in Child Process from pid 6585, iteration = 1
This line is in Child Process from pid 6585, iteration = 2
This line is in Child Process from pid 6585, iteration = 3
This line is in Child Process from pid 6585, iteration = 4
This line is in Child Process from pid 6585, iteration = 5
This line is in Child Process from pid 6585, iteration = 6
This line is in Child Process from pid 6585, iteration = 7
This line is in Child Process from pid 6585, iteration = 8
This line is in Child Process from pid 6585, iteration = 9
This line is in Child Process from pid 6585, iteration = 10
[hewll@mills ~/test/process]
```

Though parent and child processes are running in parallel, parent process usually finished earlier because there is a large overhead copying everything in parent process to create the child process, so it is likely that the system schedules parent process earlier than child process.

Set Scheduling Policy on Linux

- Function to set process priority is declared in <sched.h>

  int sched_setscheduler(pid_t **pid**, int **policy**, const struct sched_param ***param**);

where **pid** is of type pid_t is declared in <sys/types.h>

- Linux supports the following "normal" (i.e., non-real-time) scheduling policies (represented by **policy**):
  - **SCHED_OTHER**: the standard round-robin time-sharing policy;
  - **SCHED_BATCH**: for "batch" style execution of processes;
  - **SCHED_IDLE**: for running *very* low priority background jobs;
- Linux supports the following “real-time" scheduling policies:
  - **SCHED_FIFO**: a First-In-First-Out policy;
  - **SCHED_RR**: a round-robin policy.
  - **SCHED_DEADLINE**: for earliest deadline first policy.

sched_setscheduler()

```
#include <sched.h>
....
struct sched_param param;
param.sched_priority = 10;
if( sched_setscheduler(pid, SCHED_FIFO, &param ) == -1) {
        perror("sched_setscheduler failed in Parent Process.\n");
}
...
```

Need to be a privileged user to make a function call, sched_setscheduler()!!!

Code Example 2:
fork() with priority

```
#include  <stdio.h>
#include  <string.h>
#include  <sys/types.h>
#include  <sched.h>

void  main(void)
{
    pid_t  pid;
    int    i=0, num =10, ni;
    struct sched_param param;

    pid = fork();
```

```
if (pid == 0){
    pid = getpid();
    for (i = 1; i <= num; i++)
        printf("This line is in Child Process from pid %d, iteration = %d\n", pid, i);
  }
  else{
    pid = getpid();

    param.sched_priority = 18;
    if(sched_setscheduler(pid, SCHED_RR, &param) == -1)
        perror("sched_setscheduler failed in Parent Process.\n");

    ni=nice(18);
    printf("nice() function returns %d in Parent Process.\n", ni);

    for (i = 1; i <= num; i++)
        printf("This line is in Parent Process from pid %d, iteration = %d\n", pid, i);
  }
}
```

Effect of changing process priority

```
hew11@mills:~/test/process
[hew11@mills ~/test/process] ./forkPriority
This line is in Child Process from pid 23444, iteration = 1
sched_setscheduler failed in Parent Process.
This line is in Child Process from pid 23444, iteration = 2
: Operation not permitted
This line is in Child Process from pid 23444, iteration = 3
This line is in Child Process from pid 23444, iteration = 4
This line is in Child Process from pid 23444, iteration = 5
This line is in Child Process from pid 23444, iteration = 6
This line is in Child Process from pid 23444, iteration = 7
This line is in Child Process from pid 23444, iteration = 8
This line is in Child Process from pid 23444, iteration = 9
This line is in Child Process from pid 23444, iteration = 10
nice() function returns 18 in Parent Process.
This line is in Parent Process from pid 23443, iteration = 1
This line is in Parent Process from pid 23443, iteration = 2
This line is in Parent Process from pid 23443, iteration = 3
This line is in Parent Process from pid 23443, iteration = 4
This line is in Parent Process from pid 23443, iteration = 5
This line is in Parent Process from pid 23443, iteration = 6
This line is in Parent Process from pid 23443, iteration = 7
This line is in Parent Process from pid 23443, iteration = 8
This line is in Parent Process from pid 23443, iteration = 9
This line is in Parent Process from pid 23443, iteration = 10
[hew11@mills ~/test/process]
```

When nice() lowers the priority of parent process, child process get finished earlier.

Creating a Thread With a Specified Priority

- Structure `sched_param` is declared in #include <sched.h>. One of the fields is sched_priority which is used to set the process priority.

```
struct sched_param param;
param.sched_priority=20;
```

- Functions to set priority:

```
pthread_attr_getschedparam (&tattr, &param);
param.sched_priority = 10;
pthread_attr_setschedparam (&tattr, &param);
```

or

```
pthread_create(&thread, &tattr, worker function, arg);
```

Data Fields of pthread_attr_t

Type	Field
int	flags
int	stacksize
int	contentionscope
int	inheritsched
int	detachstate
int	sched
struct sched_param	param
struct timespec starttime deadline	period

Implementing Periodic Tasks

Sleep Functions

- You can use these functions sleep(), nanosleep(), clock_nanosleep() to generate periodic tasks.

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);

#include <time.h>
int nanosleep(const struct timespec *req, struct timespec *rem);

#include <time.h>
int clock_nanosleep(clockid_t clockid, int flags, const struct
timespec *request, struct timespec *remain);
```

sleep() v.s. nanosleep() & clock_nanosleep()

- They suspends the execution of the calling process/thread.
- sleep() has low resolution, nanosleep(), clock_nanosleep() has high resolutions since they use timespec to represent time

```
struct timespec {
        time_t tv_sec; /* seconds */
        long tv_nsec; /* nanoseconds [0 .. 999999999] */
};
```

nanosleep() v.s. clock_nanosleep()

clock_nanosleep() is more flexible.

Int clock_nanosleep (clockid_t clock_id, int flags,
const struct timespec *request, struct timespec * remain);

Example:

```
clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &t, NULL);
```

It suspends the execution of calling thread or process until the time value of the clock specified by clock_id reaches the absolute time specified by the time argument, or the process is terminated.

Argument of clock_nanosleep()

Clockid type takes one of the following values: CLOCK_REALTIME, CLOCK_MONOTONIC, and CLOCK_PROCESS_CPUTIME_ID.

If *flags* is 0 or **TIMER_ABSTIME**, then the value specified in *request* is interpreted as an interval or an absolute time as measured by the clock, *clock_id*.

```
struct timespec {
        time_t tv_sec; /* seconds */
        long tv_nsec; /* nanoseconds [0 .. 999999999] */
};
```

remain: if not NULL, it is remaining time to sleep when it has been interrupted by a signal before it reaches the specified sleep time.

Implementing A Periodic Task

```
#include <stdio.h>
#include <time.h>
void periodicTask(void);

int main() {
    struct timespec mytimespec;
    mytimespec.tv_sec = 0;
    mytimespec.tv_nsec = 500000000; /* 500 ms */

    while(1) {
        periodicTask();
        nanosleep(&mytimespec,NULL);
    }
    return 0;
}

void periodicTask(void) {
    printf("This would be printed periodically\n");
}
```

End