实验二、实验内容

1. 确认Linux系统的线程竞争范围、调度算法确认
2. 确认Linux 系统调度程序例子代码
3. 利用消息队列实现进程间的通信
4. Make Utility

1. 线程竞争范围

```
#include <pthread.h>
#include <stdio.h>
#define NUM THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD_SCOPE_SYSTEM");
      else
            fprintf(stderr, "Illegal scope value.\n");
   }
```

/* to be continued */

```
/* set the scheduling algorithm to PCS or SCS */
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
   /* create the threads */
      for (i = 0; i < NUM_THREADS; i++)
         pthread_create(&tid[i],&attr,runner,NULL);
   /* now join on each thread */
      for (i = 0; i < NUM_THREADS; i++)
         pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
   {
   /* do some work ... */
   pthread_exit(0);
}
```

2. Linux 调度程序

```c
#include <unistd.h>
#include <pthread.h>
#include <sched.h>

static int get_thread_policy(pthread_attr_t attr)
{
    int policy;
    pthread_attr_getschedpolicy(&attr, &policy);
    switch(policy)
    {
        case SCHED_FIFO:
            printf("policy = SCHED_FIFO\n");
            break;
        case SCHED_RR:
            printf("policy = SCHED_RR\n");
            break;
        case SCHED_OTHER:
            printf("policy = SCHED_OTHER\n");
            break;
        default:
            printf("policy = UNKOWN\n");
            break;
    }
    return policy;
}
```

```c
static void show_thread_priority(pthread_attr_t attr, int policy)
{
    int priority = sched_get_priority_max(policy);
    printf("max_priority = %d\n", priority);
    priority = sched_get_priority_min(policy);
    printf("min_priority = %d\n", priority);
}

static int get_thread_priority(pthread_attr_t attr)
{
    struct sched_param param;
    pthread_attr_getschedparam(&attr, &param);
    printf("priority = %d\n", param.sched_priority);
    return param.sched_priority;
}

static void set_thread_policy(pthread_attr_t attr, int policy )
{
    pthread_attr_setschedpolicy(&attr, policy);
    get_thread_policy(attr);
}
```

```c
int main(void)
{
    pthread_attr_t attr;
    struct sched_param sched;
    int rs = pthread_attr_init(&attr);
    int policy = get_thread_policy(attr);
    printf("- show current configuration of priority\n");
    show_thread_priority(attr, policy);
    printf("- show SCHED_FIFO of priority\n");
    show_thread_priority(attr, SCHED_FIFO);
    printf("- show SCHED_RR of priority\n");
    show_thread_priority(attr, SCHED_RR);
    printf("- show priority of current thread\n");
    int priority = get_thread_priority(attr);
    printf("SET THREAD POLICY\n");
    printf("set SCHED_FIFO policy\n");
    set_thread_policy(attr, SCHED_FIFO);
    printf("set SCHED_RR policy\n");
    set_thread_policy(attr, SCHED_RR);
    printf("restore current policy\n");
    set_thread_policy(attr, policy);
    pthread_attr_destroy(&attr);
    return 0;
}
```

3. 消息队列进程间的通信

利用消息队列，实现进程间的通信

实验要求: 编写程序，让父进程创建两个子进程P1和P2，并使子进程P1和P2通过消息队列相互通信，发送消息（512字节），基于下面代码进行修改

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <stropts.h>
#include <time.h>
#include <strings.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

```
struct msg {
        long msg_types;
        char msg_buf[511];
};
```

```
int main (void) {
        int qid, pid, len;
        struct msg pmsg;
        sprintf ( pmsg.msg_buf, “hello! this is :%d\n\0”, getpid());
        len = strlen ( pmsg.msg_buf);
        if ( (qid = msgget( IPC_PRIVATE, IPC_CREAT | 0666)) < 0 ) {
                perror( “msgget”);
                exit (1);
        }
        if ( (msgsnd(qid, &pmsg, len, 0)) < 0) {
                perror( “msgsnd”);
                exit(1);
        }
        printf(“successfully send a message to the queue: %d\n”, qid);
        exit(1);
}
```

4. Make Utility

什么是 Make Utility?

- 一般项目由几百或几千个源文件组成，编译是一项复杂而繁琐的工作，比如，对源程序部分文件进行修改后，重新编译很费时间。解决的问题的方法就是make utility。

- Make Utility 是一个命令解释工具，它解释配置文件中的指令（规则）
- Make Utility 可以只针对被修改的源文件进行重新编译

Example

```
/* File name : main.c */
#include “a.h”
…
```

```
/* File name : 2.c */
#include “a.h”
#include “b.h”
…
```

```
/* File name : 3.c */
#include “b.h”
#include “c.h”
…
```

如修改了 c.h 文件，因 main.c 和 2.c 文件没有依赖关系，无需重新编译 main.c 和 2.c 文件

Make Utility 命令选项

- -k : Keep-going, continue as much as possible after an error
- -n : Build-test, print the commands that would be executed, but do not execute.
- -f <filename> : use filename as a makefile
- Others, you can use “$man make” command

What is makefile?

通过编辑 makefile 配置文件，简化编译工作。makefile 文件中主要内容如下：

1. 编译配置
2. 编译时编译规则
3. 编译后对生成文件的管理和配置

What is makefile?

- 定义了一系列的规则来指定哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至进行更复杂的功能操作，因为 Makefile 就像一个 Shell Script 一样，其中也可以执行操作系统的命令。
- 定义了源文件编译过程中，编译后，以及生成文件的存放规则等
- Makefile 文件一般存放在源文件的根目录

Makefile Format

1. 指定了依赖关系

- 指定生成的目标文件与源文件的依赖关系

2. 指定了生成规则

- 指定从源文件生成目标文件的生成规则
- 以<Tab> 开始

Makefile Example

```
/* Filename : Makefile1 */
myapp: main.o 2.o 3.o
        gcc –o myapp main.o 2.o 3.o

main.o: main.c a.h
        gcc –c main.c
2.o: 2.c a.h b.h
        gcc –c 2.c
3.o: 3.c b.h c.h
        gcc –c 3.c
```

- 最终可执行文件 myapp 与 main.o, 2.o, 3.o 文件有依赖性
- Main.o 目标文件 与 main.c, a.h 文件有依赖关系
- 2.o 目标文件 与 2.c, a.h, b.h 文件有依赖关系
- 3.o 目标文件 与 3.c, b.h, c.h 文件有依赖关系
- 目标文件 2.o 文件 创建规则是 gcc -c 2.c

Example

```
/* Filename : b.h */
#include <stdio.h>

void function_two();
```

```
/* Filename : c.h */
#include <stdio.h>

void function_three();
```

```
/*Filename:  2.c */
#include “a.h”
#include “b.h”

void function_two() {
        printf(“ This is 2\n”);
}
```

```
/ * Filename : 3.c */
#include “a.h”
#include “b.h”

void function_three() {
        printf(“ This is 3\n”);
}
```

Example

```
/* Filename : a.h */

#include <stdio.h>

void function_two();
void function_three();
```

```
/* Filename : main.c */
#include “a.h”

1. extern void function_two();
2. extern void function_three();

3. int main()
4. {
5.          function_two();
6.          function_three();
7.          return 0;
8. }
```

Example

- 运行

```
$ make -f makefile1
    gcc -c main.c
    gcc -c 2.c
    gcc -c 3.c
    gcc -o myapp main.o 2.o 3.o
$
```

Example

修改 b.h 文件以后， 重新运行 make

```
$ make -f makefile1
gcc -c 2.c
gcc -c 3.c
gcc -o myapp main.o 2.o 3.o
$
```

Example

把 object 文件删除后，重新执行 make

```
$ rm 2.o
$ make -f Makefile1
gcc -c 2.c
gcc -o myapp main.o 2.o 3.o
$
```

Makefile 宏指令

- 更一般的形式
- 可以指定编译选项

Define:

MACRONAME = value

Usage:

$(MACRONAME) or ${MACRONAME}

Makefile 宏指令

```
/* File name : Makefile2 */
all: myapp

# Which compiler
CC = gcc
# Where are include files kept
INCLUDE = .
# Options for development
CFLAGS = -g -Wall -ansi
# Options for release
# CFLAGS = -O -Wall -ansi
```

-g : 可调试模式

-O :对代码进行基本优化

-Wall :设置警告

-ansi:C 标准编译

-c :只编译，不连接

Makefile 宏指令

```
myapp: main.o 2.o 3.o
   $(CC) –o myapp main.o 2.o 3.o

main.o: main.c a.h
   $(CC)  –I$(INCLUDE)  $(CFLAGS) –c main.c

2.o: 2.c a.h b.h
   $(CC)  –I$(INCLUDE)  $(CFLAGS) –c 2.c

3.o: 3.c b.h c.h
    $(CC)  –I$(INCLUDE)  $(CFLAGS) –c 3.c
```

Makefile 宏指令

```
$ rm *.o myapp
$ make -f makefile2
    gcc -I. -g -Wall -ansi -c main.c
    gcc -I. -g -Wall -ansi -c 2.c
    gcc -I. -g -Wall -ansi -c 3.c
    gcc -o myapp main.o 2.o 3.o
$
```

Others

通过编译生成的目标文件、库文件、可执行文件等需要方便去管理，

- 如库文件存放在 /usr/lib 或 /lib 目录下
- 可执行文件存放在 /bin 目录下
- 临时生成的 object 文件管理

$make config

- 配置编译环境

$make clean

- 删除临时生成的目标文件

$make Install

- 把可执行文件移动/复制到相应的目录下

Q&A