Thread Level Scheduling

- The process scheduler schedules only the kernel threads.
- User threads are mapped to kernel threads by the thread library - The OS
- Two types of Contention Scope in Scheduling
- ***Contention scope*** refers to the scope in which threads compete for the use of physical CPUs –
- On systems implementing many-to-one and many-to-many threads ***Process Contention Scope, PCS,*** occurs, because competition occurs between threads that are part of the same process

- ***System Contention Scope, SCS,***

involves the system scheduler scheduling kernel threads to run on one or more CPUs.

- PTHREAD_SCOPE_PROCESS schedules threads using PCS = using the many-to-many model.
- PTHREAD_SCOPE_SYSTEM schedules threads using SC - effectively implementing a one-to-one model
- getscope and setscope methods provide for determining and setting the scope contention respectively:

```
#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");
    }
```

```
   /* 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);
}
```

POSIX Support for Scheduling

- POSIX defines two scheduling classes for real time threads, SCHED_FIFO and SCHED_RR, depending on how threads ***of equal priority*** share time.
- SCHED_FIFO schedules tasks in a first-in-first-out order, with no time slicing among threads of equal priority.
- SCHED_RR performs round-robin time slicing among threads of equal priority.
- POSIX provides methods for getting and setting the thread scheduling policy

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[])
{
   int i, policy;
   pthread_t tid[NUM_THREADS];
   pthread_attr_t attr;

   /* get the default attributes */
   pthread_attr_init(&attr);

   /* get the current scheduling policy */
   if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
      fprintf(stderr, "Unable to get policy.\n");
   else {
      if (policy == SCHED_OTHER)
       printf("SCHED_OTHER\n");
      else if (policy == SCHED_RR)
       printf("SCHED_RR\n");
      else if (policy == SCHED_FIFO)
       printf("SCHED_FIFO\n");
   }
```

```
   /* set the scheduling policy - FIFO, RR, or OTHER */
   if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
      fprintf(stderr, "Unable to set policy.\n");

   /* 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);
}
```

Signal Handling in Linux

- Broad classification - synchronous and asynchronous
- A signal is an asynchronous event which is delivered to a process. •
- Asynchronous means that the event can occur at any time may be unrelated to the execution of the process. •
- Signals are raised by some error conditions, such as memory segment violations, floating point processor errors, or illegal instructions. – e.g. user types ctrl-C, or the modem hangs

Signal Handling in Linux

- A signal – an event which is generated to notify a process or thread that some important situation has risen
- a process or thread on receipt of a signal, will stop what its doing and take some action
- Can be seen as a form of inter-process communication.
- signals are defined in the header file *signal.h* as a macro constant
- Every signal has a name and an associated number. Advisable to track on Signal names to avoid changing numbers on different systems

Signal Name	Description
SIGHUP	Hang-up the process. The SIGHUP signal is used to report disconnection of the user's terminal, possibly because a remote connection is lost or hangs up.
SIGINT	Interrupt the process. When the user types the INTR character (normally Ctrl + C) the SIGINT signal is sent.
SIGQUIT	Quit the process. When the user types the QUIT character (normally Ctrl + \) the SIGQUIT signal is sent.
SIGILL	Illegal instruction. When an attempt is made to execute garbage or privileged instruction, the SIGILL signal is generated. Also, SIGILL can be generated when the stack overflows, or when the system has trouble running a signal handler.

SIGTRAP	Trace trap. A breakpoint instruction and other trap instruction will generate the SIGTRAP signal. The debugger uses this signal.
SIGABRT	Abort. The SIGABRT signal is generated when abort() function is called. This signal indicates an error that is detected by the program itself and reported by the abort() function call.
SIGFPE	Floating-point exception. When a fatal arithmetic error occurred the SIGFPE signal is generated.
SIGUSR1 and SIGUSR2	The signals SIGUSR1 and SIGUSR2 may be used as you wish. It is useful to write a signal handler for them in the program that receives the signal for simple inter-process communication.

Default Action Of Signals

- Each signal has a default action, one of the following:
- **Term:** The process will terminate.

  Core: The process will terminate and produce a core dump file.

  Ign: The process will ignore the signal.

  Stop: The process will stop.

  Cont: The process will continue from where it stopped.
- Default action may be changed using handler function. Some signal's default action cannot be changed.
- **SIGKILL** and **SIGABRT** signal's default action cannot be changed or ignored.