Lab 7 pthreads

Intro =>Threads are often described as light-weight processes. They will work independently like processes but can share the same global variables. Thread operations include thread creation, termination, synchronization (joins,blocking), scheduling, data management and process interaction. A thread does not maintain a list of created threads, nor does it know the thread that created it.

All threads within a process share the same address space.

Threads in the same process share:

1. Process instructions
2. Most data
3. open files (descriptors)
4. signals and signal handlers
5. current working directory
6. User and group id

Each thread has a unique:

1. Thread ID
2. set of registers, stack pointer
3. stack for local variables, return addresses
4. signal mask
5. priority
6. Return value: errno
7. pthread functions return "0" if OK.

PTHREAD SYSTEM CALLS

==========

Pthread_create() => For creating threads If successful it return 0 otherwise it generates a nonzero number.

code in pthread library for pthread create =>

int pthread_create(

pthread_t *threaded, //id of thread data type that holds information about threads

const pthread_attr_t *attr, //attributes of threadt holds details about the thread,like scheduling policy, stack size etc

void *(*start_routine) (void*), //function that is to assign its the function the thread executes. The function needs to have a void pointer as argument and must return a void* pointer (void* can be interpreted as a pointer to anything)

void *arg //pointer to argument of function. To pass multiple arguments, send a pointer to a structure.

);

==========

Pthread_join() => Wait of thread termination and join if successful it return 0 otherwise it generates a nonzero number.

code in pthread library for pthread join =>

```
Int pthread_join (
Pthread_t threaded,    //id of thread which have to join
void **retval //return status of thread
);
```

==========

EXAMPLE OF CREATION AND JOINING:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

struct args {
    char* name;
    int age;
};

void *hello(void *input) {
    printf("name: %s\n", ((struct args*)input)->name);
    printf("age: %d\n", ((struct args*)input)->age);
}

int main() {
    struct args *Allen = (struct args *)malloc(sizeof(struct args));
    char allen[] = "Allen";
    Allen->name = allen;
    Allen->age = 20;
    pthread_t tid;
    pthread_create(&tid, NULL, hello, (void *)Allen);
    pthread_join(tid, NULL); }
```

SOME PTHREAD ATTRIBUTES

==========

pthread_attr_init() => Initializes a thread attributes object attr with the default value. if successful completion, it will return a 0 otherwise, an error number is returned to indicate the error.

Prototype => **int pthread_attr_init(pthread attr t *attr)**

example:

```
#include <pthread.h>
pthread_attr_t tattr;
int ret;
/* initialize an attribute to the default value */
ret = pthread_attr_init(&tattr);
```

==========

pthread_attr_setdetachstate() => controls whether the thread is created in a detached state. Thread state is detached means it cannot be joined with other threads.

Prototype => **int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate)**

PTHEAD_CREATE_DETACHEDT => Thread state is detached means it cannot be joined with other threads.

PTHREAD_CREATE_JOINABLE => Thread state is join able means it can be joined with other threads

example:

```
#include <pthread.h>
pthread_attr_t tattr;
int Ret;
/* set the thread detach state */
Ret=pthread_attr_setdetachstate(&tattr,PTHREAD_CREATE_DETACHED);
```

==========

pthread_attr_destroy() => Controls detach state of a thread Destroys attribute objects
Prototype => **int pthread_attr_destroy(pthread_attr_t *attr)**

example:

```
#include <pthread.h>
pthread_attr_t tattr;
int ret;

/* destroy an attribute */
ret = pthread_attr_destroy(&tattr);
```

==========

EXAMPLE : CREATING A DETACHABLE THREAD:

```
#include <pthread.h>

pthread_attr_t tattr;
pthread_t tid;
void *start_routine;
void arg
int ret;

/* initialized with default attributes */
ret = pthread_attr_init()(&tattr);
ret = pthread_attr_setdetachstate()(&tattr,PTHREAD_CREATE_DETACHED);
ret = pthread_create()(&tid, &tattr, start_routine, arg);
```

==========

```
=========================================================================

#include<math.h>
#include<pthread.h>
#include<stdlib.h>

long double x,fact[150], pwr[150],s[1];
int i,term;

void *Power(void *temp) {
     int k;
     for(k=0;k<150;k++) {
          pwr[k] = pow(x,k);
     //printf("%.2Lf\n",pwr[k]);
}
     return pwr;
}

void *Fact(void *temp) {
     long double f;
     int j;
     fact[0] = 1.0;
     for(term=1;term<150;term++) {
          f = 1.0;
          for(j=term;j>0;j--)
               f = f * j;
          fact[term] = f;
          //printf("%.2Lf\n",fact[term]);
     }
     return fact;
}

void *Exp(void *temp) {
     int t;
     s[0] = 0;
     for(t=0;t<150;t++)
          s[0] = s[0] + (pwr[t] / fact[t]);
     return s;
}
```

```
int main(void) {

    pthread_t thread1,thread2,thread3;
    long double **sum;

    printf("Exponential [PROMPT] Enter the value of x (between 0 to
    100) (for calculating exp(x)):");

    scanf("%Lf",&x);

    printf("\nExponential [INFO] Threads creating.....\n");

    pthread_create(&thread1,NULL,Power,NULL); //calling power
    function
    pthread_create(&thread2,NULL,Fact, NULL); //calling factorial
    function

    printf("Exponential [INFO] Threads created\n");

    pthread_join(thread1,NULL);
    pthread_join(thread2,NULL);

    printf("Exponential [INFO] Master thread and terminated threads
    are joining\n");

    printf("Exponential [INFO] Result collected in Master thread\n");
    pthread_create(&thread3,NULL,Exp,NULL);
    pthread_join(thread3,sum);

    printf("\neXPONENTIAL [INFO] Value of exp(%.2Lf)
    is : %Lf\n\n",x,s[0]);
    exit(1);

}
```

lAB 8 OPENMP

OpenMP Is: An Application Program Interface (API) that may be used to explicitly direct multithreaded,
shared memory parallelism. Comprised of three primary API components:

- Compiler Directives
- Runtime Library Routines
- Environment Variables

OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization. Parallelization can be as simple as taking a serial program and inserting compiler directives.... Or as complex as inserting subroutines to set multiple levels of parallelism, locks and
even nested locks.

==========

Compiler Directives:

Compiler directives appear as comments in your source code and are ignored by compilers unless you tell them
OpenMP compiler directives are used for various purposes:

- Spawning a parallel region
- Dividing blocks of code among threads
- Distributing loop iterations between threads
- Serializing sections of code
- Synchronization of work among threads

syntax: #pragma omp directive-name [clause, ...]
For example: #pragma omp parallel default(shared) private(beta,pi)

==========

Run-time Library Routines:
The OpenMP API includes an ever-growing number of run-time library routines.
These routines are used for a variety of purposes:

- Setting and querying the number of threads
- Querying a thread's unique identifier (thread ID), a thread's ancestor's identifier, the
thread team size

• Setting and querying the dynamic threads feature
• Querying if in a parallel region, and at what level
==========

syntax:

```
#include <omp.h>
omp_set_num_threads()
omp_get_num_threads()
omp_get_max_threads()
omp_get_thread_num()
omp_get_num_procs()
omp_init_lock()
omp_destroy_lock()
omp_set_lock()
omp_unset_lock()
omp_test_lock()
```

==========

Environment Variables:

OpenMP provides several environment variables for controlling the execution of parallel code at run-time.

These environment variables can be used to control such things as:

• Setting the number of threads
• Specifying how loop iterations are divided
• Binding threads to processors
• Enabling/disabling nested parallelism; setting the maximum levels of nested
parallelism
• Enabling/disabling dynamic threads
• Setting thread stack size
• Setting thread wait policy

Setting OpenMP environment variables is done the same way you set any other environment variables,
and depends upon which shell you use. For example:
syntax:

```
export OMP_NUM_THREADS=8
OMP_SCHEDULE
```

OMP_NUM_THREADS
OMP_DYNAMIC
OMP_NESTED
==========

EXAMPLE:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main (int argc, char *argv[]) {
int i, tid, nthreads, n = 10, N = 100000000;
double *A, *B, tResult, fResult;
time_t start, stop;
clock_t ticks;
long count;
A = (double *) malloc(N*sizeof(double));
B = (double *) malloc(N*sizeof(double));
for (i=0; i<N; i++) {
A[i] = (double)(i+1);
B[i] = (double)(i+1);
}
time(&start);
//this block use single process
for (i=0; i<N; i++)
{
fResult = fResult + A[i] + B[i];
}
//begin of parallel section
#pragma omp parallel private(tid, i,tResult) shared(n,A,B,fResult)
{
tid = omp_get_thread_num();
if (tid == 0) {
nthreads = omp_get_num_threads();
printf("Number of threads = %d\n", nthreads);
}
#pragma omp for schedule (static, n)
for (i=0; i < N; i++) {
tResult = tResult + A[i] + B[i];
}
#pragma omp for nowait
for (i=0; i < n; i++)
{
printf("Thread %d does iteration %d\n", tid, i);
```

```
}
#pragma omp critical
fResult = fResult + tResult;
}
//end of parallel section
time(&stop);
printf("%f\n",fResult);
printf("Finished in about %.0f seconds. \n", difftime(stop, start));
exit(0);
}
```

==========

Clauses:

For general attributes:

Clause	Description
if	Specifies whether a loop should be executed in parallel or in serial.
num_threads	Sets the number of threads in a thread team. ordered Required on a parallel for statement if an ordered directive is to be used in the loop.
schedule	Applies to the for directive.
nowait	Overrides the barrier implicit in a directive.

==========

For data-sharing attributes:

Clause	Description
private	Specifies that each thread should have its own instance of a variable.
shared	Specifies that one or more variables should be shared among all threads.
default	Specifies the behavior of unscoped variables in a parallel region.

==========

Lab 9 Semaphores:

There are two types of locks:

1. **Shared lock** (used for reading purposes)
2. **Exclusive lock** (used for writing purposes)

With **shared lock** more than one reading processes can be in critical section of the code where they access the shared resource, whereas in **exclusive lock**, not more than a single process is allowed to access the resource.

Semaphores: A semaphore is a counter that can be used to synchronize multiple threads. Linux guarantees that checking or modifying the value of a semaphore can be done safely, without creating a race condition. Each semaphore has a counter value, which is a non-negative integer.

Two basic operations:

A **wait** operation decrements the value of the semaphore by 1. If the value is already zero, the operation blocks until the value of the semaphore becomes positive (due to the action of some other thread). When the semaphore's value becomes positive, it is decremented by 1 and the wait operation returns.

A **post** operation increments the value of the semaphore by 1. If the semaphore was previously zero and other threads are blocked in a wait operation on that semaphore, one of those threads is unblocked and its wait operation completes (which brings the semaphore's value back to zero).

Two types of semaphores:

Named semaphores are powerful semaphores that are usually created to share among processes running from different files. This type of semaphores creates a temporary file under the directory /temp, that stores the value of semaphore.

Unnamed semaphores are less powerful semaphores that often work in a single file, they have no presence outside the file.

A semaphore is represented by a variable whose type is **sem_t** and is defined in **semaphore.h** library

The **sem_init()** initializes an unnamed semaphore Returns 0 if success otherwise -1

Prototype: int sem_init(sem_t *sem, int pshared, unsigned int value)

sem specifies the semaphore to be initialized.
pshared indicates whether the semaphore is shared among the processes (pshared is zero the semaphore is not shared otherwise shared)
value specifies the value to assign to the newly initialized semaphore.

Example:

```
sem_t sem;
sem_init(&sem, 0, 1);
```

if the value of semaphore is greater than zero, **sem_wait()** will immediately decreases it by 1 and returns. Otherwise the process is blocked. This function returns 0 if success and -1 when error.

Prototype: int sem_wait(sem_t *sem)

sem_post() Increments the value of the semaphore by 1 pointed by the sem_t. If the value becomes greater than zero the process is unblocked. Returns 0 on success and -1 on error.

Prototype: int sem_post(sem_t *sem)

Example:

```
Sem_t sem;
if(sem_wait(&sem)) {
//some critical stuff
}
//after critical stuff
sem_post(&sem);
```

sem_destroy() Destroys an unnamed semaphore pointed by the variable sem_t, if there are process waiting for this semaphore or the semaphore is already destroyed. The program may produce an undefined behavior. This function returns 0 if success or -1 if error.

Prototype : int sem_destroy(sem_t *sem)

Example:

```
sem_t sem;
sem_destroy(&sem);
```

lab 10 SIGNALS:

Signals inform processes of the occurrence of asynchronous events.

events called **interrupts**, as they must interrupt the regular flow of a program in order to be processed. When UNIX recognizes that such an event has occurred, it sends the corresponding process a signal. The kernel isn't the only one that can send a signal; any process can send any other process a signal, as long as it has permissions.

A programmer may arrange for a particular signal to be ignored or to be processed by a special piece of code called a **signal handler**. In the latter case, the process that receives the signal suspends its current flow of control, executes the signal handler, and then resumes the original flow of control when the signal handler finishes.

Signals inform processes of the occurrence of asynchronous events. Every type of signal has a handler which is a function. All signals have default handlers which may be replaced with user-defined handlers. The default signal handlers for each process usually terminate the process or ignore the signal, but this is not always the case.

The default handler usually performs one of the following actions:

- terminates the process and generates a core file (dump) terminates
- the process without generating a core image file (quit) ignores and
- discards the signal (ignore)
- suspends the process (suspend)
- resumes the process

SIGHUP HUP is a short form of "hang up." Locate the terminal to be controlled or hung up on the death of the control process. This signal is received when the process runs from the terminal, and that terminal goes abruptly.

SIGINT (polite kill) is sent to every process associated with a control terminal when the interrupt key (Control-C) is hit. It is used to kill a process

SIGSTOP (sleep signal) is Sent to every process associated with a control terminal when the interrupt key (Control-Z) is hit. It is used to suspend a process by sending it the signal, which is like a sleep signal, that can be undone and the process can be resumed again.

SIGQUIT signal is issued when the user sends the quite signal (Ctrl + D or ctrl + \). Commonly sent in order to get acore dump.

SIGILL used for illegal instruction. Sent when the hardware detects an illegal instruction. Sometimes a process using floating point aborts with this signal when it is accidentally linked without the -f option on the cc command.

SIGTRAP (Trace trap/ debugger) Sent after every instruction when a process is run with tracing turned on with ‘ptrace’.

SIGIOT (I/O trap instruction) Sent when a hardware fault occurs, the exact nature of which is up to the implementer and is machine dependent.

SIGTERM (Software termination) The standard termination signal. It's the default signal sent by the kill command, and is also used during system shutdown to terminate all active processes. A program should be coded to either let this signal default oand call exit.

SIGKILL (Kill) The one and only sure way to kill a process, since this signal is always fatal To be used only in emergencies; SIGTERM is preferred.

SIGEMT (Emulator trap instruction) is sent when an implementation dependent hardware fault occurs. Extremely rare.

SIGFPE (Floating-point exception) is sent when the hardware detects a floating-point error, such as a floating point number with an illegal format. Almost always indicates a program bug.

SIGBUS (Bus error) Sent when an implementation-dependent hardware fault occurs. Usually means that the process referenced at an odd address data that should have been word aligned.

SIGSEGV (Segmentation violation) means that the process referenced data outside its address space. Trying to use NULL pointers will usually give you a SIGSEGV.

SIGPIPE Write on a pipe not opened for reading. Sent to a process when it writes on a pipe that has no reader. Usually this means that the reader was another process that terminated abnormally. This signal acts to terminate all processes in a pipeline: When a process terminates abnormally, all processes to its right receive an end-of-file and all processes to its left receive this signal.

SIGALRM (Alarm clock)is sent when a process's alarm clock goes off. The alarm clock is set with the alarm system call.

SIGUSR1 (User defined signal 1) This signal may be used by application programs for interprocess communication. This is not recommended however, and consequently this signal is rarely used.

SIGUSR2 User defined signal 2. Similar to SIGUSR1.

SIGPWR Power-fail restart. Exact meaning is implementation-dependent. One possibility is for it to be sent when power is about to fail (voltage has passed, say, 200 volts and is falling). The process has a very brief time to execute. It should normally clean up and exit (as with SIGTERM). If the process wishes to survive the failure (which might only be a momentary voltage drop), it can clean up and then sleep for a few seconds. If it wakes up it can assume that the disaster was only a dream and resume processing. If it doesn't wake up, no further action is necessary. Programs that need to clean up before terminating should arrange to catch signals SIGHUP, SIGINT, and SIGTERM. Until the program is solid, SIGQUIT should be left alone so there will be a way to terminate the program (with a core dump) from the keyboard. Arrangements for the other signals are made much less often; usually they are left to terminate the process. But a really polished program will want to catch everything it can, to clean up, possibly log error, and print a nice error message. Psychologically, a message like “internal error 53: contact customer support'' is more acceptable than the message ``Bus error – core dumped’’ from the shell. For some signals, the default action of termination is accompanied by a core dump. These are SIGQUIT, SIGILL, SIGTRAP, SIGIOT, SIGEMT, SIGFPE, SIGBUS, SIGSEGV, and SIGSYS.
int alarm (int count)

alarm() instructs the kernel to send the SIGALRM signal to the calling process after count seconds. If an alarm had already been scheduled, it is overwritten. If count is 0, any pending alarm requests are cancelled. **alarm()** returns the number of seconds that remain until the alarm signal is sent.

EXAMPLE:

```
#include <stdio.h>
main ( )
{
alarm (5) ; /* schedule an alarm signal in 5 seconds */
printf ("Looping forever ...\n") ;
while ( 1 ) ;
printf ("This line should never be executed.\n") ;
}
```

signal System Call

The first argument, sig, is a signal number. The second argument, func, can be one of three things:

• **SIG_DFL**. This sets the default action for the signal.

• **SIG_IGN**. This sets the signal to be ignored; the process becomes immune to it. The signal SIGKILL can't be ignored. Generally, only SIGHUP, SIGINT, and SIGQUIT should ever be permanently ignored. The receipt of other signals should at least be logged, since they indicate that something exceptional has occurred.

• A pointer to a function. This arranges to catch the signal; every signal but SIGKILL may becaught. The function is called when the signal arrives.

Example:

```
    #include <signal.h>
    signal(sig, func) /* Catch signal with func */
    void (func)();/* The function to catch the sig
    /*Returns the previous handler or -1 on error */
```

pause System Call

pause() suspends the calling process and returns when the calling process receives a signal. It is most often used to wait efficiently for an alarm signal. pause() doesn't return anything useful.

The following program catches and processes the SIGALRM signal efficiently by having user written signal handler, alarmHandler (), by using signal ().

```
#include <stdio.h>
#include <signal.h>
int alarmFlag = 0 ;
void alarmHandler ( ) ;
int main ( ) {
signal(SIGALRM, alarmHandler) ; /*Install signal Handler*/
alarm (5) ;
printf ("Looping ...\n") ;
while (!alarmFlag) {
```

```
pause ( ); /* wait for a signal */
printf ("Loop ends due to alarm signal\n");
}
return 0;}
void alarmHandler ( ) {
printf ("An ALARM clock signal was received\n");
alarmFlag = 1;
}
```

Ha! Ha!

```
#include <stdio.h>
#include <signal.h>
void haha(int sig);
int main()
{
int i;
signal(SIGINT, haha);
for (i = 1; i <= 2000000000; i++)
{
//do nothing
}
return 0;
}
void haha(int sig)
{
printf("Ha! Ha! \n");
}
```