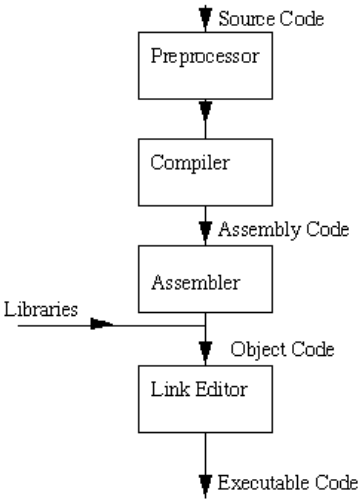
Workshop 3

18th March 2021

Subject: COMP30023

Tutor: Priyankar Bhattacharjee

The code compilation process



Creating a thread:

Initially your main () comprises a single, default thread.

All other threads are to be explicitly created by the programmer.

Calling pthread_create creates a new thread and makes it executable.

pthread_create arguments:

- **thread**: A unique identifier for the new thread returned by the subroutine.
- **attr**: Set thread attributes or leave NULL for system to use default values.
- **start_routine**: the routine to call once a thread is created.
- **arg**: the argument you need passed to the start_routine. It must be passed by reference as a pointer cast of type void. If not passing argument, leave NULL.

```
/* thread creation example */

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

```
void *PrintHello(void *threadid)
{
  long tid;
  tid = (long)threadid;
  printf("Hello World! It's me, thread #%ld!\n", tid);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int rc;
  long t;
  for(t=0; t<NUM_THREADS; t++){
    printf("In main: creating thread %ld\n", t);

        printf("address being passed: %p", &thread[t]);

    rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
    if (rc){
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }

  /* Last thing that main() should do */
  pthread_exit(NULL);
}
```

```
/* thread join example */

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

#define NUM_THREADS     4

void *BusyWork(void *t)
{
  int i;
  long tid;
  double result=0.0;
  tid = (long)t;
  printf("Thread %ld starting...\n",tid);
  for (i=0; i<1000000; i++)
  {
    result = result + sin(i) * tan(i);
  }
  printf("Thread %ld done. Result = %e\n",tid, result);
  pthread_exit((void*) t);
}

int main (int argc, char *argv[])
{
  pthread_t thread[NUM_THREADS];
  pthread_attr_t attr;
  int rc;
  long t;
  void *status;

  /* Initialize and set thread detached attribute */
```

```
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

  for(t=0; t<NUM_THREADS; t++) {
    printf("Main: creating thread %ld\n", t);
    rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
    if (rc) {
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
      }
    }

  /* Free attribute and wait for the other threads */
  pthread_attr_destroy(&attr);
  for(t=0; t<NUM_THREADS; t++) {
    rc = pthread_join(thread[t], &status);
    if (rc) {
      printf("ERROR; return code from pthread_join() is %d\n", rc);
      exit(-1);
      }
    printf("Main: completed join with thread %ld having a status of %ld\n",t,(long)status);
    }

printf("Main: program completed. Exiting.\n");
pthread_exit(NULL);
}
```

Resolving race condition through mutex

```
/* mutex operation demo */

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

void *functionC();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

main()
{
  int rc1, rc2;
  pthread_t thread1, thread2;

  /* Create independent threads each of which will execute functionC */

  if( (rc1=pthread_create( &thread1, NULL, &functionC, NULL)) )
  {
    printf("Thread creation failed: %d\n", rc1);
  }

  if( (rc2=pthread_create( &thread2, NULL, &functionC, NULL)) )
  {
    printf("Thread creation failed: %d\n", rc2);
  }

  /* Wait till threads are complete before main continues. Unless we  */
```

```
  /* wait we run the risk of executing an exit which will terminate   */
  /* the process and all threads before the threads have completed.   */

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

  exit(0);
}

void *functionC()
{

  pthread_mutex_lock( &mutex1 );
  counter++;
  printf("locked by thread tid: %ld\n", gettid());
  printf("Counter value: %d\n",counter);
  pthread_mutex_unlock( &mutex1 );
}
```

Suggested reading http://www.cs.kent.edu/~farrell/sp/reference/multi-thread.html#thread_mutex

Fork

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/wait.h>

/* This program forks and and the prints whether the process is
 *   - the child (the return value of fork() is 0), or
 *   - the parent (the return value of fork() is not zero)
 *
 */

int main( void ) {
      char *argv[3] = {"Command-line", ".", NULL};

      int pid = fork();

      if ( pid == 0 ) {
            execvp( "find", argv );
      }

      /* Put the parent to sleep for 2 seconds--let the child finished executing */
      wait( 2 );

      printf( "Finished executing the parent process\n"
            " - the child won't get here--you will only see this once\n" );

      return 0;
  }
```

Pipe

```c
/* example to demonstrate pipe based ipc */

#include <sys/wait.h> /* wait */
#include <stdio.h>
#include <stdlib.h>   /* exit functions */
#include <unistd.h>   /* read, write, pipe, _exit */
#include <string.h>

#define ReadEnd  0
#define WriteEnd 1

void report_and_exit(const char* msg) {
  perror(msg);
  exit(-1);   /** failure **/
}

int main() {
  int pipeFDs[2]; /* two file descriptors */
  char buf;      /* 1-byte buffer */
  const char* msg = "Nature's first green is gold\n"; /* bytes to write */

  if (pipe(pipeFDs) < 0) report_and_exit("pipeFD");
  pid_t cpid = fork();                    /* fork a child process */
  if (cpid < 0) report_and_exit("fork");          /* check for failure */

  if (0 == cpid) {   /*** child ***/             /* child process */
    close(pipeFDs[WriteEnd]);                    /* child reads, doesn't write */

    while (read(pipeFDs[ReadEnd], &buf, 1) > 0)      /* read until end of byte stream */
      write(STDOUT_FILENO, &buf, sizeof(buf));       /* echo to the standard output */

    close(pipeFDs[ReadEnd]);                     /* close the ReadEnd: all done */
    _exit(0);                                /* exit and notify parent at once  */
  }
  else {          /*** parent ***/
    close(pipeFDs[ReadEnd]);                     /* parent writes, doesn't read */

    write(pipeFDs[WriteEnd], msg, strlen(msg));      /* write the bytes to the pipe */
    close(pipeFDs[WriteEnd]);                    /* done writing: generate eof */

    wait(NULL);                              /* wait for child to exit */
    exit(0);                                /* exit normally */
  }
  return 0;
}
```

Further reading

https://hpc-tutorials.llnl.gov/posix/