Programming with fork/exec and pthreads

CIS 620 Advanced Operating Systems

Haodong Wang

1

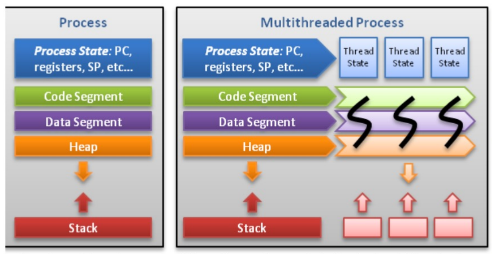
Processes

- Processes contain information about program resources and program execution state4, including:
  - Process ID, process group ID, user ID, and group ID
  - Environment, Working directory, Program instructions
  - Registers, Stack, Heap
  - Shared libraries, IPC tools (e.g., queues, pipes, semaphores, or shared memory).
- Process creation
  - Shell command: ./a.out
  - Programming level: fork() system call

2

Threads

- Threads use, and exist within, the process resources
- Scheduled and run as independent entities
- Duplicate only the bare essential resources that enable them to exist as executable code

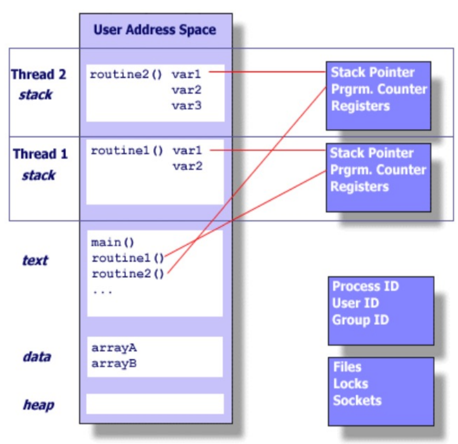


Threads contain only necessary information, such as a stack (for local variables, function arguments, return values), a copy of the registers, program counter and any thread-specific data to allow them to be scheduled individually. Other data is shared within the process between all threads.

3

Threads

- A thread maintains its own:
  - Stack pointer
  - Registers
  - Scheduling properties
  - Signals (pending or blocked)
  - Thread specific data
- Multiple threads share the process resources
- A thread dies if the process dies
- “lightweight” for creating and terminating threads that for processes



4

What are pthreads?

- Threads used to implement parallelism I shared memory multiprocessor systems, such as SMPs
- Historically, hardware vendors have implemented their own proprietary versions of threads
- For Unix, a standardized C language threads programming interface has ben specified by the IEEE Posix 1003.1c standard
  - pthreads
  - defines how threads should be created, managed, and destroyed

5

What are pthreads?

- Posix 1003.1c defines a thread interface
  - pthreads
  - defines how threads should be created, managed, and destroyed
- Unix provides a pthreads library
  - API to create and manage threads
  - you don't need to worry about the implementation details
    - this is a good thing

6

POSIX Thread API

- Commonly referred to as Pthreads, POSIX has emerged as the standard threads API, supported by most vendors.
  - Implemented with a pthread.h header/include file and a thread library
- Functionalities
  - Thread management, e.g., creation and joining
  - Thread synchronization primitives
    - Mutex
    - Conditional Variables
    - Reader/writer locks
    - Pthread barrier
  - Thread-specific data

7

PThread API

- `#include <pthread.h>`

Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects.
pthread_cond_	Condition variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data keys

- **`gcc -lpthread`**

8

Thread Creation

- Initially, main() program contains a single thread
  - All other threads must be explicitly created

```
int pthread_create(
    pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine)(void *),
    void * arg);
```

- **thread**: an opaque, unique identifier for the new thread returned by the subroutine
- **attr**: an opaque attribute object that may be used to set thread attributes, e.g., NULL
- **start_routine**: the C routing that the thread will execute once it is created
- **arg**: a single argument that may be passed to start_routine. It must be passed by reference as a pointer cast of type void. MULL may be used if no argument is to be passed.

9

Example 1. pthread_create

```
#include <pthread.h>
#define NUM_THREADS5

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

int main(int argc, char *argv[]) {
   pthread_t threads[NUM_THREADS];
   long t;

   for(t=0;t<NUM_THREADS;t++) {
     printf("In main: creating thread %ld\n", t);
     int rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t );
     if (rc) {
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
     }
   }
  pthread_exit(NULL);
}
```

```
One possible output:

In main: creating thread 0
In main: creating thread 1
In main: creating thread 2
In main: creating thread 3
Hello World! It's me, thread #0!
In main: creating thread 4
Hello World! It's me, thread #1!
Hello World! It's me, thread #3!
Hello World! It's me, thread #2!
Hello World! It's me, thread #4!
```

10

Terminating Threads

- pthread_exit is used to explicitly exit a thread
  - Called after a thread has completed its work and is no longer required to exist
- If main() finishes before the threads it has created
  - If exits with pthread_exit, the other threads will continue to execute
  - Otherwise, the will be automatically terminated when main() finishes
- The programmer may optionally specify a termination status, which is stored as a void pointer for any thread that may join the calling thread
- Cleanup: the pthread_exit() routine does not close files
  - Any file opened inside the thread will remain open after the thread is terminated

11

Thread Attribute

```
int pthread_create(
     pthread_t *thread,
     const pthread_attr_t *attr,
     void *(*start_routine)(void *),
     void * arg);
```

- Attribute contains details about
  - Whether scheduling policy is inherited or explicit
  - Scheduling policy, scheduling priority
  - Stack size, stack guard region size
- **pthread_attr_init** and **pthread_attr_destroy** are used to initialize/destroy the thread attribute object
- Other routine are then used to query/set specific attributes in the thread attribute object

12

Passing Arguments to Threads

- The pthread_create() routine permits the programmer to pass **one** argument to the thread start routine
- For cases where multiple arguments must be passed:
  - Create a structure which contains all of the arguments
  - Then pass a pointer to the object of that structure in the pthread_create() routine
  - All arguments must be passed by reference and cast to (void *)
- Make sure that all passed data is thread safe
  - It cannot be changed by other threads
  - It can be changed by a determinant way

13

Example 2: Argument Passing

```
#include <pthread.h>
#define NUM_THREADS 8

struct thread_data {
  int  thread_id;
  char *message;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg) {
  int taskid;
  char *hello_msg;

  sleep(1);
  struct thread_data *my_data = (struct thread_data *) threadarg;
  taskid = my_data->thread_id;
  hello_msg = my_data->message;
  printf("Thread %d: %s\n", taskid, hello_msg);
  pthread_exit(NULL);
}
```

14

Example 2: Argument Passing

```
int main(int argc, char *argv[]) {
   pthread_t threads[NUM_THREADS];
   int t;
   char *messages[NUM_THREADS];
   messages[0] = "English: Hello World!";
   messages[1] = "French: Bonjour, le monde!";
   messages[2] = "Spanish: Hola al mundo";
   messages[3] = "Klingon: Nuq neH!";
   messages[4] = "German: Guten Tag, Welt!";
   messages[5] = "Russian: Zdravstvytye, mir!";
   messages[6] = "Japan: Sekai e konnichiwa!";
   messages[7] = "Latin: Orbis, te saluto!";

   for(t=0;t<NUM_THREADS;t++) {
     struct thread_data * thread_arg = &thread_data_array[t];
     thread_arg->thread_id = t;
     thread_arg->message = messages[t];
     pthread_create(&threads[t], NULL, PrintHello, (void *) thread_arg);
   }
   pthread_exit(NULL);
}
```

```
Thread 3: Klingon: Nuq neH!
Thread 0: English: Hello World!
Thread 1: French: Bonjour, le monde!
Thread 2: Spanish: Hola al mundo
Thread 5: Russian: Zdravstvytye, mir!
Thread 4: German: Guten Tag, Welt!
Thread 6: Japan: Sekai e konnichiwa!
Thread 7: Latin: Orbis, te saluto!
```

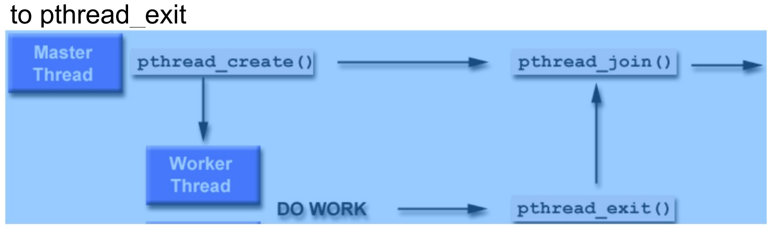
15

Wait for Thread Termination

- Suspend execution of calling thread until thread terminates

```
#include <pthread.h>
int pthread_join(
   pthread_t thread,
   void **value_ptr);
```

- Thread: the joining thread
- Value_ptr: ptr to location for return code a terminating thread passes to pthread_exit



16

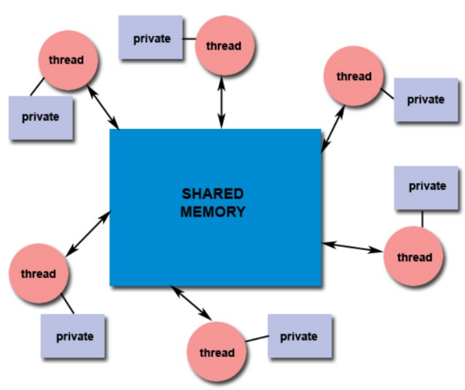
Thread Joining Example

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

void printMsg(char* msg) {
   printf(“%s\n”, msg);
}

int main(int argc, char** argv) {
   pthread_t thrdID;

   printf(“creating a new thread\n”);
   pthread_create(&thrdID, NULL, (void*)printMsg, argv[1]);
   printf(“created thread %d\n”. thrdID);
   pthread_join(thrdID, NULL);

   return 0;
}
```

17

Shared Memory and Threads

- Threads have access to the same global, shared memory
- Threads also have their own private data
- Programmers are responsible for synchronizing access globally shared data



18

Thread Caveat

- Shared State!
  - Accidental changes to global variables can be fatal
  - Changes made by one thread to shared resources (such as closing a file) will be seen by all other threads
  - Reading and writing to the same memory locations is possible
  - Therefore requires explicit synchronization by the programmer
- Many library functions are not thread-safe
  - Library functions that return pointers to static internal memory. E.g., gethostbyname()
- Lack of robustness
  - Crash in one thread will crash the entire process

19

Why Pthreads (not processes)?

- The primary motivation
  - To realize potential program performance gains
- Compared to the cost of creating and managing a process
  - A thread can be created with much less OS overhead
- Managing thread requires fewer system resources
- All threads within a process share the same address space
- Inter-thread communication is more efficient and, in many cases, easier than IPC

20

Pthread_create vs. Fork

- Timing results for the fork() system call and pthread_create() API
  - Timings reflect 50k process/thread creations
  - Units are in seconds
  - No optimization flags

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
AMD 2.4 GHz Opteron (8cpus/node)	41.07	60.08	9.01	0.66	0.19	0.43
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.24	30.78	27.68	1.75	0.69	1.10
IBM 1.5 GHz POWER4 (8cpus/node)	104.05	48.64	47.21	2.01	1.00	1.52
INTEL 2.4 GHz Xeon (2 cpus/node)	54.95	1.54	20.78	1.64	0.67	0.90
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.54	1.07	22.22	2.03	1.26	0.67

21

Synchronizing Threads

- Three basic synchronization primitives
  1. mutex locks
  2. condition variables
  3. semaphores
- Mutexes and condition variables will handle most of the cases you need in this class
  - but feel free to use semaphores if you like

22

Mutex Locks

- A Mutex lock is created like a normal variable
  – *pthread_mutex_p mutex;*
- Mutexes must be initialized before being used
  – a mutex can only be initialized once
  – prototype:
    - int pthread_mutex_init(pthread_mutex_t *mp, const pthread_mutexattr_t *mattr);
      – *mp:* a pointer to the mutex lock to be initialized
      – *mattr:* attributes of the mutex – usually NULL

23

Locking a Mutex

- To insure mutual exclusion to a critical section, a thread should lock a mutex
  – when locking function is called, it does not return until the current thread owns the lock
  – if the mutex is already locked, calling thread blocks
  – if multiple threads try to gain lock at the same time, the return order is based on priority of the threads
    - higher priorities return first
    - no guarantees about ordering between same priority threads
  – prototype:
    - int pthread_mutex_lock(pthread_mutex_t *mp);
      – *mp:* mutex to lock

24

Unlocking a Mutex

- When a thread is finished within the critical section, it needs to release the mutex
  - calling the unlock function releases the lock
  - then, any threads waiting for the lock compete to get it
  - very important to remember to release mutex
  - prototype:
    - int pthread_mutex_unlock(pthread_mutex_t *mp);
      - *mp:* mutex to unlock

25

Example

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

#define MAX_SIZE  5
pthread_mutex_t bufLock;
int count;

void producer(char* buf) {
  for(;;) {
    while(count == MAX_SIZE);
    pthread_mutex_lock(bufLock);
    buf[count] = getChar();
    count++;
    pthread_mutex_unlock(bufLock);
  }
}
```

```
void consumer(char* buf) {
  for(;;) {
    while(count == 0);
    pthread_mutex_lock(bufLock);
    useChar(buf[count-1]);
    count--;
    pthread_mutex_unlock(bufLock);
  }
}

int main() {
  char buffer[MAX_SIZE];
  pthread_t p;
  count = 0;
  pthread_mutex_init(&bufLock);
  pthread_create(&p, NULL, (void*)producer, &buffer);
  consume(&buffer);
  return 0;
}
```

26

Condition Variables (CV)

- Notice in the previous example a *spin-lock* was used wait for a condition to be true
  - the buffer to be full or empty
  - spin-locks require CPU time to run
    - waste of cycles
- Condition variables allow a thread to block until a specific condition becomes true
  - recall that a blocked process cannot be run
    - doesn’t waste CPU cycles
  - blocked thread goes to wait queue for condition
- When the condition becomes true, some other thread signals the blocked thread(s)

27

Condition Variables (CV)

- A CV is created like a normal variable
  - *pthread_cond_t condition;*
- CVs must be initialized before being used
  - a CV can only be initialized once
  - prototype:
    - int pthread_cond_init(pthread_cond_t *cv, const pthread_condattr_t *cattr);
      - *cv:* a pointer to the conditon variable to be initialized
      - *cattr:* attributes of the condition variable – usually NULL

28

Blocking on CV

- A wait call is used to block a thread on a CV
  - puts the thread on a wait queue until it gets signaled that the condition is true
    - even after signal, condition may still not be true!
  - blocked thread does not compete for CPU
  - the wait call should occur under the protection of a mutex
    - this mutex is automatically released by the wait call
    - the mutex is automatically reclaimed on return from wait call
- prototype:
  - int pthread_cond_wait(pthread_cond_t *cv,pthread_mutex_t *mutex);
    - *cv*: condition variable to block on
    - *mutex:* the mutex to release while waiting

29

Signaling a Condition

- A signal call is used to “wake up” a single thread waiting on a condition
  - multiple threads may be waiting and there is no guarantee as to which one wakes up first
  - thread to wake up does not actually wake until the lock indicated by the wait call becomes available
  - condition thread was waiting for may not be true when the thread actually gets to run again
    - should always do a wait call inside of a while loop
  - if no waiters on a condition, signaling has no effect
  - prototype:
    - int pthread_cond_signal(pthread_cond_t *cv);
      - *cv:* condition variable to signal on

30

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

#define MAX_SIZE  5
pthread_mutex_t lock;
pthread_cond_t notFull, notEmpty;
int count;

void producer(char* buf) {
  for(;;) {
    pthreads_mutex_lock(lock);
    while(count == MAX_SIZE)
          pthread_cond_wait(notFull, lock);
    buf[count] = getChar();
    count++;
    pthread_cond_signal(notEmpty);
    pthread_mutex_unlock(lock);
  }
}
```

```c
void consumer(char* buf) {
  for(;;) {
    pthread_mutex_lock(lock);
    while(count == 0)
          pthread_cond_wait(notEmpty, lock);
    useChar(buf[count-1]);
    count--;
    pthread_cond_signal(notFull);
    pthread_mutex_unlock(lock);
  }
}

int main() {
  char buffer[MAX_SIZE];
  pthread_t p;
  count = 0;
  pthread_mutex_init(&bufLock);
  pthread_cond_init(&notFull);
  pthread_cond_init(&notEmpty);
  pthread_create(&p, NULL, (void*)producer, &buffer);
  consume(&buffer);
  return 0;
}
```

31

More on Signaling Threads

- The previous example only wakes a single thread
  - not much control over which thread this is
- Perhaps all threads waiting on a condition need to be woken up
  - can do a broadcast of a signal
  - very similar to a regular signal in every other respect
- Prototype:
  - int pthread_cond_broadcast(pthread_cond_t *cv);
    - *cv:* condition variable to signal all waiters on

32