Threads and Multithreading

- Each process has an address space and a single thread of control.
- Multithreading allows a process to have other threads of execution.
- Fork is used to create a new (child) process; problems with *fork*:
  1. *fork* is too expensive mainly because memory has to be copied from parent to the child process (e.g. duplicating all descriptors)
  2. Since each process has its own address space, IPC is required to pass information between parent and child after the fork.

Reference:
https://computing.llnl.gov/tutorials/pthreads/

1

Threads solve both these problems:

- Thread creation can be 10-100 times faster than process creation.
- All threads of a process share the same global memory, making information sharing easy. Synchronization is required.

2

per **process** items:

- address space
- global variables
- open files,
- current working directory
- timers
- signals
- semaphores
- accounting information
- user and group Ids

3

per **thread** items:

- thread ID
- program counter
- stack (for local variables and return addresses)
- state
- register set
- child threads
- priority
- *errno*

Q: what do threads of same process have in common (share)?

4

Basic Posix Thread Functions

- In real-time Posix each process can contain several "threads" of execution.
- When a program is started by *exec*, a single thread is created, called the *initial thread* or the *main thread*.

5

. **Additional threads are created by** *pthread_create* **function which takes 4 pointer arguments:**

(similar to fork in process)

```
#include <pthread.h>
int   pthread_create ( pthread_t *tid,    const pthread_attr_t  *attr,
                       void * (*func) (void *),    void *arg);
```

where

- tid **is the *thread ID* (returned by the call)**
- attr **is a set of attributes. Each thread has several *attributes*: its initial stack size, its priority, whether it should be a daemon thread or not, and so on.**

  We can specify these attributes by initializing a *pthread_attr_t* variable that overrides the defaults.

  Normally the default values are taken, by specifying the attr **argument as null pointer.**
- func **is the address of thread start function. The thread starts by calling this function and then terminates either explicitly (by calling *pthread_exit*) or implicitly (by letting the function return).**
- arg **points to a set of parameters to be passed to the function when it is called.**

The return value from pthread functions is either 0 if OK or a positive number (Exxx) on an error. (for example, if pthread_create() cannot create a new thread because system limitations (on number of threads) exceeded, then return value is EAGAIN)

6

A thread can be aborted by use of pthread_cancel.

Waiting for a thread

- One thread can wait for another thread to terminate by calling the *pthread_join* function: (similar to waitpid() for process)

```
#include <pthread.h>

int    pthread_join (pthread_t tid, void **status);
```

7

```
#include <pthread.h>

int  pthread_join (pthread_t tid, void **status);
```

where

- *tid* is thread id of the thread we want to wait on
- **status*, if not null, takes the return value from the thread as a pointer to some object
  (i.e. return value is stored in the location on which status is pointing)

8

Detach

The term **detaching** means cleaning up after a thread and reclaiming its storage space.

Two ways to achieve this:

- by calling the *pthread_join* function and waiting for it to terminate, or
- by setting the detach attribute of the thread (either at creation time or dynamically by calling the *pthread_detach* function:

(a thread is either joinable or detached)

9

#include <pthread.h>

int pthread_detach (pthread_t tid);

If the detached attribute is set , the thread is not joinable and its storage space may be reclaimed automatically when the thread terminates.

When a program terminates all its threads terminate. Hence, sometimes it is necessary for a main program to issue a *pthread_join*.

10

Other thread functions

- A thread can determine its id by calling *pthread_self*:

#include <pthread.h>

pthread_t pthread_self (void);

(*similar to getpid() in process*)

11

- A thread can detach itself by calling

pthread_detach(pthread_self());

- One way for a thread to terminate is to call *pthread_exit*:

```
#include <pthread.h>
void  pthread_exit(void *status);
```

If the thread is not detached, its thread ID and exit status are retained for a later *pthread_join* by some other thread .

12

- Forking processes (programs) which contain multiple threads is not straightforward, because some threads in the process may hold resources or executing system calls.

- The **POSIX** standard specifies that the child process will only have one thread.

13

Posix mutexes and condition variables

- The thread extension to Posix provides ***mutexes*** and ***condition variables***.

- They can be used to synchronize threads of the same process.

  They can also be used to synchronize threads of multiple processes, if the mutex or condition variable is stored in memory that is shared between processes.

14

- *Mutexes* and condition variables, when combined, provide the functionality of a monitor but with a procedure interface.

  Each monitor has an associated *mutex* variable and all operations on the monitor are surrounded by calls to lock and unlock the *mutex*.

15

Mutexes: Locking and Unlocking

- Mutexes are used to provide mutually exclusive access to critical sections.
  (mutex status is locked or unlocked)
- Posix mutexs are variables declared to be of type

  pthread_mutex_t

- Statically allocated mutexes can be initialized to the constant PTHREAD_MUTEX_INITIALIZER:

```
static  pthread_mutex_t   mtxlock =
                PTHREAD_MUTEX_INITIALIZER;
```

16

- Dynamically allocated mutexes can be initialized at run time by calling the pthread_mutex_init function.
- Lock and unlock functions:

```
#include <pthread.h>

int   pthread_mutex_lock(pthread_mutex_t  *mptr);
```

* *If the mutex is locked by some other thread, then we are blocked until the mutex is unlocked.*

```
int pthread_mutex_trylock (pthread_mutex_t  *mptr);
```

* *_trylock* returns *EBUSY* if the mutex is already locked.

```
int pthread_mutex_unlock (pthread_mutex_t  *mptr);
```

17

- The normal usage:

```
pthread_mutex_lock (the_mutex);
        ……..
        Critical Section
        ……..
pthread_mutex_unlock (the_mutex);
```

18

```
#include  "pthread.h"
#define   NLOOP 5000
int                 counter;          /* this is incremented by the threads */

void        *doit(void *);

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

 Pthread_create(&tidA, NULL, &doit, NULL);
 Pthread_create(&tidB, NULL, &doit, NULL);
        /* wait for both threads to terminate */
Pthread_join(tidA, NULL);
Pthread_join(tidB, NULL);
        exit(0);
}

void *  doit(void *vptr)
{ int       i, val;
        /*
         * Each thread fetches, prints, and increments the counter NLOOP times.
         * The value of the counter should increase monotonically.
         */
        for (i = 0; i < NLOOP; i++) {
                val = counter;
                printf("%d: %d\n", pthread_self(), val + 1);
                counter = val + 1;
        }       return(NULL);
}
```

19

```
#include  "pthread.h"
#define   NLOOP 5000
int                 counter;          /* this is incremented by the threads */
pthread_mutex_t     counter_mutex = PTHREAD_MUTEX_INITIALIZER;
void        *doit(void *);

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

   Pthread_create(&tidA, NULL, &doit, NULL);
   Pthread_create(&tidB, NULL, &doit, NULL);
    /*wait for both threads to terminate */
  Pthread_join(tidA, NULL);
  Pthread_join(tidB, NULL);
        exit(0);
}

void  * doit(void *vptr)
{  int              i, val;
        / * Each thread fetches, prints, and increments the counter NLOOP times.
         * The value of the counter should increase monotonically. */
 for (i = 0; i < NLOOP; i++) {
        Pthread_mutex_lock(&counter_mutex);
        val = counter;
        printf("%d: %d\n", pthread_self(), val + 1);
        counter = val + 1;
        Pthread_mutex_unlock(&counter_mutex);
        }
        return(NULL);
}
```

20

Example:

The Multiple-Producers/Single-Consumer Problem:

- Producers and consumer are represented by threads,
- buff is an integer array to hold the shared data,
- producers set buff[i] to i,
- consumer verifies that each entry is correct,
- consumer starts after all producers terminate.

21

Example: mutex-producer-consumer

```
#include    "unpipc.h"
#define     MAXNITEMS          1000
#define     MAXNTHREADS        100

/* globals shared by threads */
int         nitems;            /* read-only by
producer and consumer */
struct {
  pthread_mutex_t       mutex;
  int       buff[MAXNITEMS];
  int       nput; /* next index to store */
  int       nval; /* next value to store */
} shared = { PTHREAD_MUTEX_INITIALIZER };

/* end globals */
```

22

```
void  *produce(void *), *consume(void *);

/* include main */

int   main(int argc, char **argv)
{
  int        i, nthreads, count[MAXNTHREADS];
  pthread_t  tid_produce[MAXNTHREADS], tid_consume;

if (argc != 3)
   err_quit("usage: prodcons <#items> <#threads>");
nitems = min(atoi(argv[1]), MAXNITEMS);
nthreads = min(atoi(argv[2]), MAXNTHREADS);
```

23

```
        Set_concurrency(nthreads + 1);
        /* in Solaris 2.6: Thr_SetConcurrency(nthreads+1)

        /* create all producers and one consumer */
        for (i = 0; i < nthreads; i++) {
               count[i] = 0;
               pthread_create(&tid_produce[i], NULL,
                      produce, &count[i]);
        }


        /* wait for all producers */
        for (i = 0; i < nthreads; i++) {
               pthread_join(tid_produce[i], NULL);
               printf("count[%d] = %d\n", i, count[i]);

        }

        Pthread_create(&tid_consume, NULL, consume,NULL);
        Pthread_join(tid_consume, NULL);

        exit(0);
}
/* end main */
```

24

The produce and consume functions are

```
void * produce(void *arg)
{
      for ( ; ; ) {
            Pthread_mutex_lock(&shared.mutex);
            if (shared.nput >= nitems) {
            /*** buffer is full ***/
             pthread_mutex_unlock(&shared.mutex);
             return(NULL);/* we're done */
            } /** if **/
           /*** buffer is not full ***/
           shared.buff[shared.nput] = shared.nval;
           shared.nput++;
           shared.nval++;
           pthread_mutex_unlock(&shared.mutex);
           *((int *) arg) += 1;
      } /** for **/
}
```

25

```
void * consume(void *arg)
{
      int        i;

  for (i = 0; i < nitems; i++) {
      if (shared.buff[i] != i)
       printf("buff[%d] = %d\n", i,
                          shared.buff[i]);
      }
      return(NULL);
}
/* end prodcons */
```

26

Condition Variables: Waiting and Signaling

- A mutex is for locking and a condition variable is for waiting.
- A condition variable is a variable of type:
  `pthread_cond_t`
- The associated functions are:

```
#include <pthread.h>
int  pthread_cond_wait(pthread_cond_t *cptr,
                       pthread_mutex_t *mptr);

int pthread_cond_signal( pthread_cond_t *cptr);
```

(both return zero on success, nonzero on error)

27

A condition variable is always associated with a mutex.

The condition variable is a signaling mechanism associated with a mutex.

When a thread waits on a condition variable, its lock on the associated mutex is released.

When the thread successfully returns from the wait, it again holds the lock.

However, because more than one thread may be released, the programmer must again test for the condition which caused it to wait initially.

28

General structure of the code that waits on a condition variable:

```
struct {
        pthread_mutex_t      mutex;
        pthread_cond_t       cond;
        /* variables for maintaining the condition */
        } varx = {
                  PTHREAD_MUTEX_INITIALIZER,
                  PTHREAD_COND_INITIALIZER,...
                 };

        pthread_mutex_lock(&varx.mutex);
        while (condition is false)
          pthread_mutex_wait(&varx.cond, &varx.mutex);
          ... modify condition ...
        pthread_mutex_unlock(&varx.mutex);
```

29

General structure of the code that signals a condition variable:

```
pthread_mutex_lock(&var.mutex);
        set condition true
pthread_cond_signal(&var.cond);
pthread_mutex_unlock(&var.mutex);
```

The producers/consumer example where the consumer thread starts right after all producer threads have started:

The consumer must synchronize with the producers.

A non-busy-waiting solution.

30

Example: producer consumer

```
#include      "unpipc.h"
#define       MAXNITEMS           1000000
#define       MAXNTHREADS         100

/* globals shared by threads */

int           nitems;    /* read-only by producer and consumer */
int           buff[MAXNITEMS];
int           Nsignals;
struct {
  pthread_mutex_t          mutex;
  int         nput;  /* next index to store */
  int         nval;  /* next value to store */
} put = { PTHREAD_MUTEX_INITIALIZER };
  /** struct put is used by producer only ***/
 struct{
   pthread_mutex_t    mutex;
   pthread_cond_t     cond;
   int                nready  /* number ready for consumer */
} nready = {PTHREAD_MUTEX_INITIALIZER,
                     PTHREAD_COND_INITIALIZER,0  };
```

31

The main function is:

```
int main(int argc, char **argv)
{
  int               i, nthreads, count[MAXNTHREADS];
  pthread_t         tid_produce[MAXNTHREADS],  tid_consume;
  if (argc != 3)
      err_quit("usage: prodcons7 <#items> <#threads>");
  nitems = min(atoi(argv[1]), MAXNITEMS);
  nthreads = min(atoi(argv[2]), MAXNTHREADS);

  Set_concurrency(nthreads + 1);

          /* create all producers and one consumer */

for (i = 0; i < nthreads; i++) {
    count[i] = 0;
    Pthread_create(&tid_produce[i], NULL, produce, &count[i]);
}
pthread_create(&tid_consume, NULL, consume, NULL);
```

32

```
        /* wait for all producers and the consumer*/

        for (i = 0; i < nthreads; i++) {
                Pthread_join(tid_produce[i], NULL);
                printf("count[%d] = %d\n", i, count[i]);
        }
        Pthread_join(tid_consume, NULL);
        printf("nsignals = %d\n", Nsignals);

        exit(0);
}
```

33

The produce function is:

```
void *produce(void *arg)
{
  for ( ; ; )
{
        Pthread_mutex_lock(&put.mutex);
        if (put.nput >= nitems) {
                Pthread_mutex_unlock(&put.mutex);
                return(NULL); /* array is full, we're done */
        }
        buff[put.nput] = put.nval;
        put.nput++;
        put.nval++;
        Pthread_mutex_unlock(&put.mutex);

        Pthread_mutex_lock(&nready.mutex);
        If (nready.nready == 0) {
                Pthread_cond_signal(&nready.cond);
                Nsignals++;
        };
       nready.nready++;
      pthread_mutex_unlock(&nready.mutex);

 *((int *) arg) += 1;
}
}
```

34

The consume function is:

```
void *consume(void *arg)
{
  int             i;

  for (i = 0; i < nitems; i++) {
        Pthread_mutex_lock(nready.mutex);
        While (nready.nready == 0)
              Pthread_cond_wait(&nready.cond,&nready.mutex);
         nready.nready--;
         Pthread_mutex_unlock(&nready.mutex);

        if (buff[i] != i)
            printf("buff[%d] = %d\n", i, buff[i]);
        }

        return(NULL);
}
```

35

Additional functions

```
#include <pthread.h>

int  pthread_cond_broadcast(pthread_cond_t  *cptr,);
It wakes up all threads that are blocked on the condition variable.

int pthread_cond_timedwait( pthread_cond_t     *cptr,
                            pthread_mutex_t   *mutex,
                            const struct timespec  *abstime);

abstime is a timespec structure:
      struct timespec{
                time_t      tv_sec;      /* seconds */
                long        tv_nsec;     /* nanosecond */
               };
```

The structure specifies the system time (absolute time) when the function must return, even if the condition variable has not been signaled yet.

36