Threads vs. Processes

- Typically processes connote *heavyweight* things, threads *lightweight* ones
- Processes, e.g. in UNIX, contain much baggage:
  - page table
  - file descriptor table
  - processor state
  - resource tables, etc.

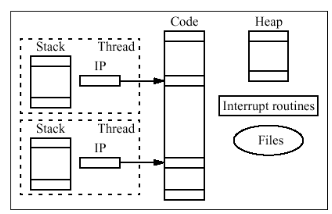
Threads vs. Processes

- Threads concentrate only on the processor state.
- Consequently, threads can be switched much more quickly.
- This provides opportunities of latency-hiding for memory access and i/o.

Threads vs. Processes

- Threads typically share logical memory within a process.
- Processes typically do not share logical memory, except for special shareable segments.
- shmalloc = “shared memory allocate”, kind of an after-thought

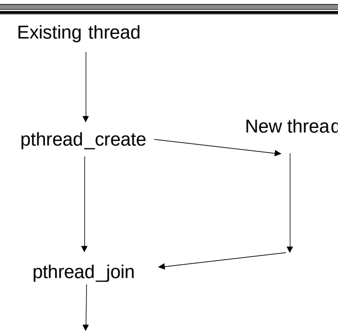
Threads within one Process



Pthreads (Posix Threads)

- Posix = an API standard, for a variety of system aspects (threads, real-time, etc.)
- Posix = “Portable UNIX”

Thread Creation and Joining



pthread_create and _exit

- pthread_create(pthread_t &tid, // thread id
  NULL, // attributes
  (void*)threadCode(void*), // code
  (void*) parameter); // params

- creates new pthread running threadCode; parameter is passed to threadCode

- pthread_exit((void*) value)

- terminates thread, passing value if joined to another thread

- pthread_join(pthread_t tid, // thread id
  (void**) result);

pthread_exit

- pthread_exit((void*) value)

- terminates thread, passing value if joined to another thread

- Note: storage for result must be allocated dynamically or outside of the thread code.

pthread_join

- pthread_join(pthread_t tid, // thread id
  (void**) result);

- waits for thread tid, result is that sent by _exit

pthread1.c example

```
struct package
{
char* msg;
};

void* threadCode(void* arg)
  {
  struct package *realArg = arg;
  printf("Hello from %s.\n", realArg->msg);
  pthread_exit(realArg->msg);
  }
```

pthread1.c example

```
struct package
{
char* msg;
};

void* threadCode(void* arg)
  {
  struct package *realArg = arg;
  printf("Hello from %s.\n", realArg->msg);
  pthread_exit(realArg->msg);
  }
```

```
int main(int argc, char** argv)
  {
  struct package arg1, arg2;
  char *result;
  pthread_t tid1, tid2;

  arg1.msg = "thread1";
  arg2.msg = "thread2";

  pthread_create(&tid1, NULL, threadCode, &arg1);
  pthread_create(&tid2, NULL, threadCode, &arg2);

  printf("Hello from main.\n");

  pthread_join(tid1, (void*)&result);
  printf("Thread 1 joined, result is %s.\n", result);

  pthread_join(tid2, (void*)&result);
  printf("Thread 2 joined, result is %s.\n", result);
  }
```

pthread1.c example

```
struct package
{
char* msg;
};

void* threadCode(void* arg)
  {
  struct package *realArg = arg;
  printf("Hello from %s.\n", realArg->msg);
  pthread_exit(realArg->msg);
  }
```

output

```
Hello from main.
Hello from thread1.
Hello from thread2.
Thread 1 joined, result is thread1.
Thread 2 joined, result is thread2.
```

```
int main(int argc, char** argv)
  {
  struct package arg1, arg2;
  char *result;
  pthread_t tid1, tid2;

  arg1.msg = "thread1";
  arg2.msg = "thread2";

  pthread_create(&tid1, NULL, threadCode, &arg1);
  pthread_create(&tid2, NULL, threadCode, &arg2);

  printf("Hello from main.\n");

  pthread_join(tid1, (void*)&result);
  printf("Thread 1 joined, result is %s.\n", result);

  pthread_join(tid2, (void*)&result);
  printf("Thread 2 joined, result is %s.\n", result);
  }
```

Exercise

- Describe how you would implement matrix multiply using pthreads.

Thread Safety

- Some library routines might not be “thread safe”.
- This is typically because they are not “reentrant”, i.e. they assume certain fixed memory locations rather than allocate all of their storage individually.

Thread Locking (non-busy)

```
// global
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
…

// in competing threads
pthread_mutex_lock(&mutex);

… critical section …

pthread_mutex_unlock(&mutex);
```

pthread2.c example

```
struct package
{
char* msg;
pthread_mutex_t* mutex;
};


/* Using a mutex below, we should never see the hello and goodbye of two
 * threads interleaved.
 */

void* threadCode(void* arg)
  {
  struct package *realArg = arg;
  pthread_mutex_lock(realArg->mutex);
  printf("Hello from %s.\n", realArg->msg);
  sleep(1);
  printf("Goodbye from %s.\n", realArg->msg);
  pthread_mutex_unlock(realArg->mutex);
  pthread_exit(realArg->msg);
  }
```

pthread2.c example

```
int main(int argc, char** argv)
  {
  pthread_mutex_t mutex1;
  struct package arg1, arg2;
  char *result;
  pthread_t tid1, tid2;

  pthread_mutex_init(&mutex1, NULL);
  arg1.msg = "thread1";
  arg2.msg = "thread2";
  arg1.mutex = &mutex1;
  arg2.mutex = &mutex1;        // one mutex is shared with both threads

  pthread_create(&tid1, NULL, threadCode, &arg1);
  pthread_create(&tid2, NULL, threadCode, &arg2);

  pthread_mutex_lock(&mutex1);
  printf("Hello from main.\n");
  sleep(1);
  printf("Goodbye from main.\n");
  pthread_mutex_unlock(&mutex1);

  pthread_join(tid1, (void*)&result);
  printf("Thread 1 joined, result is %s.\n", result);

  pthread_join(tid2, (void*)&result);
  printf("Thread 2 joined, result is %s.\n", result);
  }
```

output

```
Hello from main.
Goodbye from main.
Hello from thread1.
Goodbye from thread1.
Hello from thread2.
Thread 1 joined, result is thread1.
Goodbye from thread2.
Thread 2 joined, result is thread2.
```

Condition Variables

- Condition variables allow one thread to signal another.

Condition Variables

```
// global
pthread_cond_t cond;
pthread_cond_init(&cond, NULL);
…

// in separate threads
pthread_cond_wait(&cond, &mutex);

pthread_cond_signal(&cond);    // 1-1 signaling
pthread_cond_broadcast(&cond); // avalanche
```

Condition Variables

```
// global
pthread_cond_t cond;
pthread_cond_init(&cond, NULL);
…

// in separate threads
pthread_cond_wait(&cond, &mutex);

pthread_cond_signal(&cond);    // 1-1 signaling
pthread_cond_broadcast(&cond); // avalanche
```

Why is this here?

pthread3.c example

```
void* threadCode(void* arg)
  {
  struct package *realArg = arg;
  printf("Hello from %s.\n", realArg->msg);
  if( !strcmp(realArg->msg, "thread1") )
    {
    sleep(1);
    printf("Signalling in %s.\n", realArg->msg);
    pthread_cond_signal(realArg->cond);
    }
  else
    {
    printf("Waiting in %s.\n", realArg->msg);
    pthread_cond_wait(realArg->cond, realArg->mutex);
    printf("No longer waiting in %s.\n", realArg->msg);
    sleep(1);
    }
  printf("Goodbye from %s.\n", realArg->msg);
  pthread_exit(realArg->msg);
  }
```

```
struct package
{
char* msg;
pthread_cond_t* cond;
pthread_mutex_t* mutex;
};
```

pthread3.c example

```
int main(int argc, char** argv)
  {
  pthread_cond_t cond1;
  pthread_mutex_t mutex1;
  struct package arg1, arg2;
  char *result;
  pthread_t tid1, tid2;

  pthread_cond_init(&cond1, NULL);
  arg1.msg = "thread1";
  arg2.msg = "thread2";
  arg1.cond = &cond1;
  arg2.cond = &cond1;          // one cond is shared with both threads
  arg1.mutex = &mutex1;
  arg2.mutex = &mutex1;        // one mutex is shared with both threads

  pthread_create(&tid1, NULL, threadCode, &arg1);
  pthread_create(&tid2, NULL, threadCode, &arg2);

  printf("Hello from main.\n");
  sleep(1);
  printf("Goodbye from main.\n");

  pthread_join(tid1, (void*)&result);
  printf("Thread 1 joined, result is %s.\n", result);

  pthread_join(tid2, (void*)&result);
  printf("Thread 2 joined, result is %s.\n", result);
```

output

```
Hello from main.
Hello from thread1.
Hello from thread2.
Waiting in thread2.
Goodbye from main.
Signalling in thread1.
Goodbye from thread1.
No longer waiting in thread2.
Thread 1 joined, result is thread1.
Goodbye from thread2.
Thread 2 joined, result is thread2.
```

Major, Major Caveat

- From the man page:
  - The pthread_cond_signal() and pthread_cond_broadcast() functions have *no effect if there are no threads currently blocked on cond*.
- This means that a collection of threads may well exhibit time-dependent behavior when using this primitive.

Semaphores

- Semaphores are a better alternative to conditional variables
- They don't lose signals that may have occurred before the wait statement.
- Exactly one wait is enabled per every signal.
- Unfortunately, they are not part of Posix

Semaphores

- Each semaphore has an associated count, initially 0 by default. (May be set at > 0)
- Invariant:
  count > 0 → no processes waiting
  count = number of wait operations before blocking
- Behavior:
  - wait, or P, or down:
    if(count > 0) count--; else wait on queue;
  - signal, or V, or up:
    if(queue non-empty)
    wakeup one on queue;
    else count++;

Exercise

- Implement a semaphore data type using mutexes and conditional variables.

Semaphores implemented using pthreads

```
/* file:   bksem.h
 * author:  keller
 * purpose: Bob Keller's semaphores implemented using pthread primitives
 *
 * Declare semaphore as:
 *    struct bksem s;
 *
 * Initialize with
 *    init(&s, value);
 * where value should be non-negative.
 *
 * Operation up, signal, or V:
 *    up(&s);
 *
 * Operation down, wait, or P:
 *    down(&s)
 */
```

Semaphores implemented using pthreads

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

typedef struct
  {
  pthread_mutex_t mutex;
  pthread_cond_t  cond;
  int value;
  } bksem;

/* workings:
 *
 * The value of the semaphore maintains the following invariant, assuming
 * that the initial value is non-negative, which it always should be:
 *
 *   If the value is <= 0, this is the number of threads waiting on the
 *   the semaphore (i.e. on the associated condition variable).
 *
 *   If the value is >= 0, this is the number of times threads can perform
 *   the down operation without waiting.
 */
```

Semaphores implemented using pthreads

```
void init(bksem* s, int value)
 {
 s->value = value;
 }

void up(bksem* s)
 {
 pthread_mutex_lock(&(s->mutex));
 s->value++;
 if( s->value <= 0 )
   {
   pthread_cond_signal(&(s->cond));
   }
 pthread_mutex_unlock(&(s->mutex));
 }
```

```
void down(bksem* s)
 {
 pthread_mutex_lock(&(s->mutex));
 s->value--;
 if( s->value < 0 )
   {
   pthread_cond_wait(&(s->cond), &(s->mutex));
   }
 pthread_mutex_unlock(&(s->mutex));
 }
```

Test Program

```
/* struct representing shared data */

typedef struct
  {
  int consumerDelay;
  int producerDelay;
  bksem supply;
  bksem space;
  bksem mutex;
  int occupied;
  int vacant;
  } sharedData;

sharedData pkg;
```

Test Program

```
void* producer(void* arg)
  {
  int i;
  for( i = 0; i < CYCLES; i++ )
    {
    sleep(pkg.producerDelay);            // producer delay
    down(&pkg.space);                    // wait for space
    down(&pkg.mutex);                    // lock data
    pkg.occupied++;                      // simulate production
    pkg.vacant--;                        // reduce space
    printf("producer produces %d, occupied = %d, vacant = %d \n",
           i, pkg.occupied, pkg.vacant);
    up(&pkg.mutex);                      // unlock data
    up(&pkg.supply);                     // indicate production
    }
  }
```

Test Program

```
void* consumer(void* arg)
  {
  int i;
  for( i = 0; i < CYCLES; i++ )
    {
    down(&pkg.supply);                   // wait for supply
    sleep(pkg.consumerDelay);            // consumer delay
    down(&pkg.mutex);                    // lock data
    pkg.occupied--;                      // simulate consumption
    pkg.vacant++;                        // increase space
    printf("consumer consumes cycle %d, occupied = %d, vacant = %d
           i, pkg.occupied, pkg.vacant);
    up(&pkg.mutex);                      // unlock data
    up(&pkg.space);                      // indicate consumption
    }
  }
```

Exercise

- Implement a barrier synchronization mechanism for Posix threads.