Operating system concepts

Concepts vs Implementation

POSIX Threads

Concept vs real implementations

- implementation behavior?
- extensions?
- restrictions?

Presentation overview

- POSIX Threads
  - introduction
  - creating, ending, managing
  - thread private data

- Synchronization
  - mutex, conditional variables
  - read-write locks, barrier, spinlocks
  - POSIX semaphores
  - UNIX semaphores

POSIX

- Before POSIX:
  - many vendor standards – every UNIX distribution had own interface – wasn't portable!!!

- ***P**ortable **O**perating **S**ystem **I**nterface [for Uni**x**]*

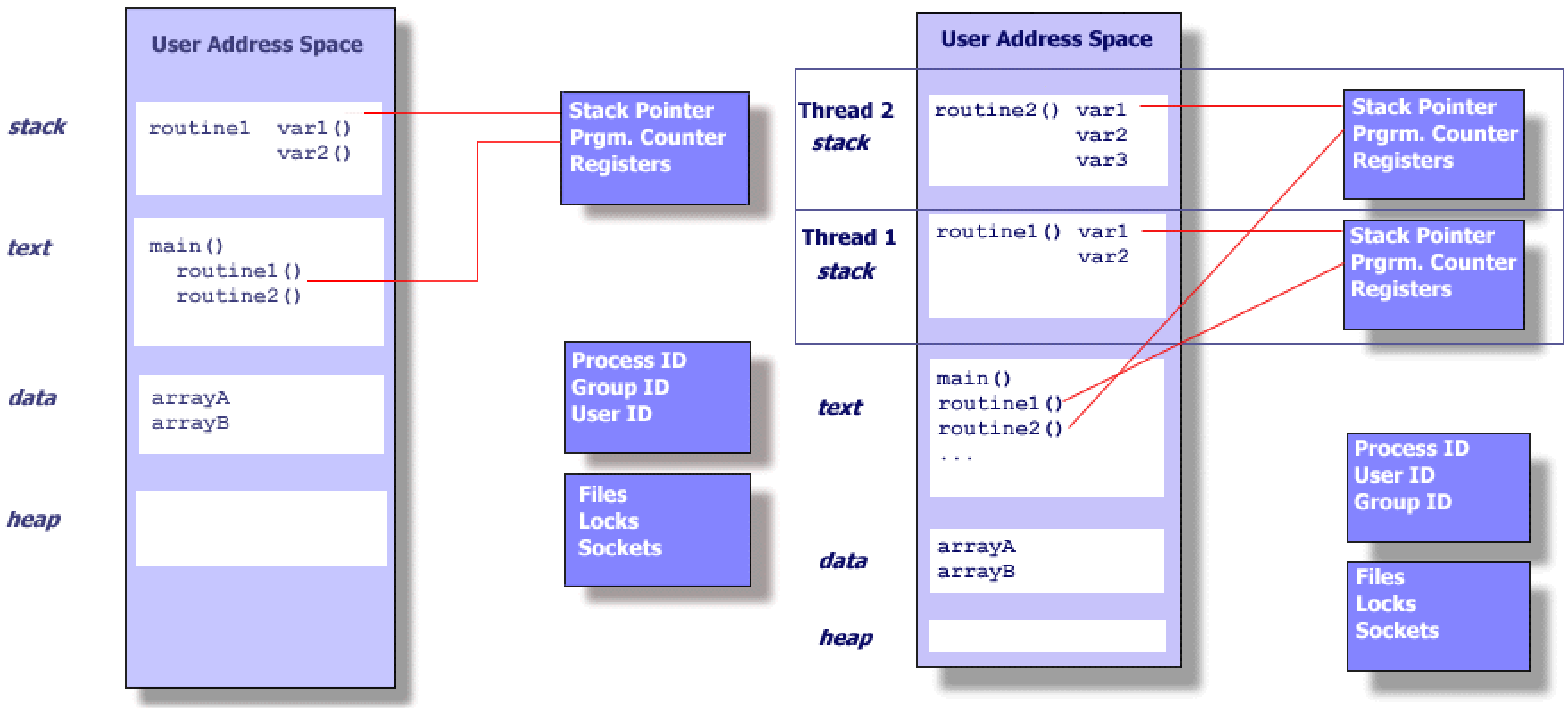
- A family of related standards specified by the IEEE to define the application programming interface (API), …
- fundamental POSIX interfaces are functionally similar to other OS interfaces (e.g. Win32)
- Emerged from a project that began ~1985
- IEEE Std 1003, ISO/IEC 9945
- http://en.wikipedia.org/wiki/POSIX (short overview)
- http://www.unix.org/2008edition
- http://www.opengroup.org

POSIX Threads

- POSIX **defines** interfaces
  - e.g. http://www.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html
- **Implementations** may not (need not to) implement all interfaces or functionality
- Why threads in focus, and not processes?

Thread vs Process: system resources

- New thread in current process use far less system resources than new process



Process & single thread Process & multiple threads

Thread vs Process: creation times

- Creation timings: 50,000 process/thread creation time (in seconds, source: https://computing.llnl.gov/tutorials/pthreads/)

Platform	fork()			pthread_ create()		
	real	user	sys	real	user	sys
AMD 2.4 GHz Opteron (8cpus/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

Thread vs Process

- Threads are "light-weight"
  - fewer OS resources
  - significantly lower overhead for thread creation
  - faster context switching
  - easiest and faster inter-thread communication
  - shared data (all process address space)
    - unprotected changes may be fatal !!!

- Threads are less secure
  - one thread may crash whole process
  - if threads work on different tasks and their work is not connected, they could be in their own processes
    - example: Google Chrome creates new process for each "tab" – each Web page is rendered in different process; when we Web page is closed – all data related to it is automatically released; if one crashes – others will not

Creating threads

- With process creation, main (first, initial) thread is created
- From C programmer perspective:
  - main thread starts with "**`main`**" function

```
int main (…)
{
  …
}
```

- If required, all other threads must be explicitly created by existing thread(s)!
- POSIX interface for thread creation: **`pthread_create`**

`pthread_create` - parameters

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

- **`thread_id`** – address where to store *handle* of created thread
- **`attr`** – various attributes for thread creation (priority/scheduling, stack, …)

- **`start_routine`** – starting routine for created thread (like “main” for first thread)
- **`arg`** – only argument passed to starting routine

Creating thread – example (pthread_create.c)

```
#include <pthread.h>
#include <stdio.h>
void *new_thread (void *p) {
   int *n = p;
   int num = *n;
   printf("In thread %d\n", num);
   return p; //or pthread_exit(p);
}
int main () {
   pthread_t t1, t2;
   int n1 = 1, n2 = 5, *status1, *status2;
   pthread_create(&t1, NULL, new_thread, (void *) &n1);
   pthread_create(&t2, NULL, new_thread, (void *) &n2);

   pthread_join(t1, (void *) &status1); //wait till thread t1 ends
   pthread_join(t2, (void *) &status2);
   printf("Collected status: %d %d\n", *status1, *status2);
   return 0;
}
```

Compiling:

```
#gcc e1.c -lpthread
#./a.out
In thread 1
In thread 5
Collected status: 1 5
#
```

Parameter to starting routine

Returning value to calling thread

Returning value to calling process (usually command shell)

Passing parameters to new thread

- Don't pass address of loop variable!

```
for (i = 0; i < N; i++)
     pthread_create(&t, NULL, thr_func, (void *) &i);
```

- Loop variable changes and intended value is not sent (usually all threads get **N**)!
- Pass value (if it is integer – address is also a number! ☺):

```
for (i = 0; i < N; i++)
     pthread_create(&t, NULL, thr_func, (void *) i);
```

- In thread function get value:

```
void * thr_func (void *p) {
   int num = (int) p;
   ...
```

- If more parameters are required, put them into structure, e.g.:

```
struct params { int a, b, c; double d; … } p[N];
for (i = 0; i < N; i++) {
     //initialize p[i] with data for thread 'i'
     pthread_create(&t[i],NULL,thr_func, (void *)&p[i]);
}
```

Managing created threads

- Created thread ends its execution ("voluntarily") with:
  - exiting from its starting function
  - calling to **`pthread_exit`**
    - e.g. thread is not in starting function

- Parent thread (or any other) may wait for thread end with
  - **`pthread_join`**

- Thread can ("forcefully") terminate other thread:
  - sending a signal to that thread
    - **`pthread_kill (thread, signal)`**
    - and in thread handling function **`pthread_exit`** is called
  - request for thread cancelation
    - **`pthread_cancel (thread)`**

- <u>If main thread end – process ends (with all its threads)!!!</u>

Reusing thread resources

- Resources are reserved for every created thread:
  - thread descriptor in kernel data structures
  - stack and private data in process address space
- When thread ends, its resources are not always released automatically!
- They are released when:
  - **`pthread_join`** is performed on them (by other thread), or
  - thread is marked as "*detachable*"
    - with **`attr`** at thread creation or later with **`pthread_detach`**
    - when *detachable* thread ends, its resources are automatically reclaimed
    - the detached thread can act as a daemon thread (while the main thread performs other operations – <u>process must exist</u>!)

Example: threads in Web server

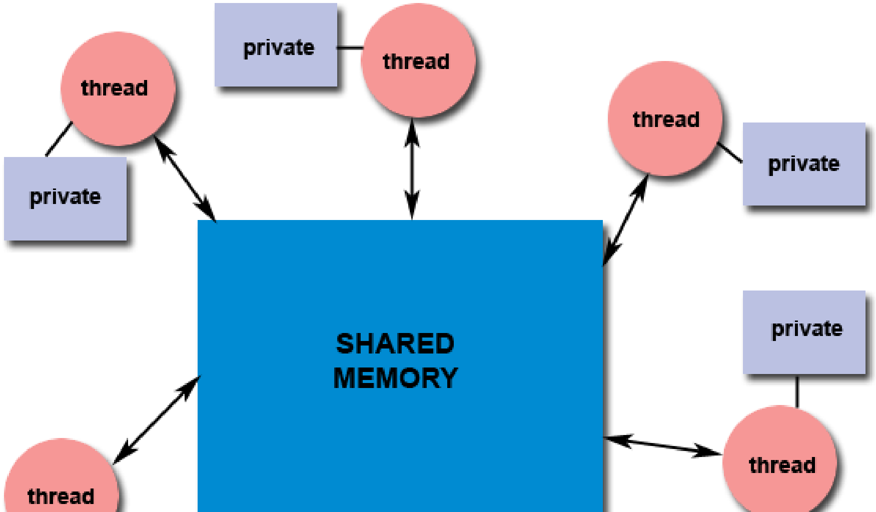
```
int main() {
   //initialization in main thread that waits on connections
   pthread_attr_t attr;
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
   //with this flag, upon finishing their processing, thread resources will be freed
   ...
   //main loop – waiting on connections and forwarding them to processing threads
   while (not_end) {
      client = malloc(sizeof(struct Client));
      client->socket = accept(srv, &client->addr, &client->len);
      pthread_create(&tid,&attr, service_thread, (void *)client);
   }
   ...
}

//service thread function
void *service_thread (void *p) {
   struct Client *client = p;
   //service this client request
   ...
}
```

Private thread data

- "Per-thread" user data – accessible for general purposes
  - for storing problem related data

- When do we need "private" thread data?
  - principle is similar to global variables, but available only to single / specific thread
  - reduce parameter number/size when calling functions
  - use when parameters can't be sent, e.g. processing of asynchronous events like signals



Private thread data – example (pthread_specific.c)

```
#include <pthread.h>
pthread_key_t thr_stat, thr_buffer;//global variables, shared among threads
...
//main thread – initialization of 'keys', basis for thread specific data
pthread_key_create(&thr_stat, free_data);
pthread_key_create(&thr_buffer, free_data);
//initially, value NULL is associated with each key for all threads
...
pthread_create...
...
//worker thread - initialization
stat = malloc(sizeof(struct ThrStat));          //stat - local variable
buffer = malloc(sizeof(struct ThrBuffer));      //buffer - local variable

//associate stat with key thr_stat for current thread only!!!
pthread_setspecific(thr_stat, stat);
pthread_setspecific(thr_buffer, buffer);
...
//worker thread – in some function
s = pthread_getspecific(thr_stat); //get data associated with key thr_stat
b = pthread_getspecific(thr_buffer);
... //use 's' and 'b'
```

```
void *free_data (void *d) {
    if (d != NULL)
        free(d);
    return NULL;
}
```

Synchronization

- Available mechanisms:
  - Mutex:
    - pthread_mutex_lock/unlock/init
  - Conditional variable
    - pthread_cond_wait/signal/broadcast/init
  - Reader/Writer lock
  - Barrier
  - Spin lock
  - Semaphore (Real-time extension)
  - UNIX semaphore
- Demonstration through examples

Monitor example – Old Bridge problem

- Old bridge (over river) puts restrictions on traffic:
  at all times:
  - cars can drive over bridge only in same direction
    - bridge is too narrow
  - no more than three cars may be crossing it
    - bridge construction is fragile (old)
- Simulate cars with threads
  - synchronize threads with monitor (mutex and cond. var.)
  - simulation must preserve given restrictions
- In following implementation bridge state is described with
  - number of cars currently on bridge - **`cars_on_bridge`**
  - direction of cars on bridge - **`dir_on_bridge`**

Old Bridge – solution (Old_bridge.c)

```
void *car_thread (void *p) {
   struct CarInfo *car = p;
   pthread_mutex_lock(&m);
   while(cars_on_bridge > 2 || (dir_on_bridge!=-1 && dir_on_bridge != car->dir))
      pthread_cond_wait(cq[car->dir], &m);
   //go on bridge
   cars_on_bridge++;
   dir_on_bridge = car->dir;
   pthread_mutex_unlock(&m);
   //drive over bridge
   usleep(5000000);//sleep 5 seconds
   //drive off bridge
   pthread_mutex_lock(&m);
   cars_on_bridge--;
   if (cars_on_bridge > 0) {
      pthread_cond_signal(cq[car->dir]);
   }
   else {
      dir_on_bridge = -1;
      pthread_cond_broadcast(cq[1-car->dir]);
   }
   pthread_mutex_unlock(&m);
   free(car);
}
```

```
int main () {
   pthread_attr_t attr;
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(
   &attr, PTHREAD_CREATE_DETACHED);
   while (1) {
      car = malloc(sizeof(struct CarInfo));
      car->id = ++car_id;
      car->dir = rand() & 1;
      pthread_create(&thr_id, &attr,
            car_thread, (void *) car);
      usleep(2000000);
   }
   return 0;
}
```

This solution is not very fair!
Fairness would require counting crosses.

Reader/Writer locks (ReaderWriter.c)

```
void *reader (int p) {
  while (1) {
    pthread_rwlock_rdlock(&rwlock);
    num_readers++;
    usleep(2000000);
    num_readers--;
    pthread_rwlock_unlock(&rwlock);
    usleep(1000000 * p);
  }
}
void *writer (int p) {
  while (1) {
    pthread_rwlock_wrlock(&rwlock);
    num_writters++;
    usleep(1000000);
    num_writters--;
    pthread_rwlock_unlock(&rwlock);
    usleep(1000000 * p * 5);
  }
}
```

Reader/writer locking principles:

- When reader acquire lock, only readers can pass through locking
- When writer locks, nor reader nor writer will pass
- When writer is waiting, no more readers are allowed to lock - even if reader thread is currently owning the lock

Barrier (Barrier.c)

```
pthread_barrier_t barrier;
void *thread (int p) {
   while (1) {
        usleep(1000000 * p);
        at_barrier++;
        pthread_barrier_wait(&barrier);
        at_barrier--;
        if(!at_barrier)
                printf("---Barrier passed---\n");
        usleep(1000000 * p);
   }
}
int main () {
   ...
   pthread_barrier_init(&barrier, NULL, 5);
   pthread_create(&thr_id, &attr, thread, (void *) 1);
   ...
   pthread_create(&thr_id, &attr, thread, (void *) 5);
   usleep(50000000); //simulation time
   return 0;
}
```

- Barrier will block threads until all threads come to barrier
  - when last thread comes to barrier - all threads are released and barrier is reset
- At barrier initialization number of threads must be provided

Spinlock (Spinlock.c)

Look at CPU usage! (high)

```
pthread_spinlock_t lock;
void *thread (int p) {
   while (1) {
        printf("Thread %d ready\n", p);
        pthread_spin_lock(&lock);
        printf("Thread %d inside C.S.\n", p);
        usleep(1000000 * p);
        printf("Thread %d leaving C.S.\n", p);
        pthread_spin_unlock(&lock);
        usleep(1000000);
   }
}
int main () {
   pthread_t thr_id;
   pthread_spin_init(&lock, PTHREAD_PROCESS_PRIVATE);
   pthread_create(&thr_id, NULL, thread, (void *) 1);
   ...
   pthread_create(&thr_id, NULL, thread, (void *) 6);
   usleep(50000000); //simulation time
   return 0;
}
```

Semaphore (Semaphore.c)

Look at CPU usage! (low)

```
#include <semaphore.h> //POSIX RT extension
sem_t sem;
void *thread (int p) {
   while (1) {
        printf("Thread %d ready\n", p);
        sem_wait(&sem);
        printf("Thread %d inside C.S.\n", p);
        usleep(1000000 * p);
        printf("Thread %d leaving C.S.\n", p);
        sem_post(&sem);
        usleep(1000000);
   }
}
int main () {
   pthread_t thr_id;
   sem_init(&sem, 0, 1);
   pthread_create(&thr_id, NULL, thread, (void *) 1);
   ...
   pthread_create(&thr_id, NULL, thread, (void *) 6);
   usleep(50000000); //simulation time
   return 0;
}
```

Initial semaphore value

UNIX semaphore (Sys_sem.c)

```
#include <sys/sem.h>
int sem;
void *thread (int p) {
   struct sembuf op;
   op.sem_num = 0;//first semaphore in set
   op.sem_flg = 0;
   while (1) {
        op.sem_op = -1; //decrement semaphore value => SemWait
        semop(sem, &op, 1);
        printf("Thread %d inside C.S.\n", p);
        usleep(1000000 * p);
        printf("Thread %d leaving C.S.\n", p);
        op.sem_op = 1; //increment semaphore value => SemPost
        semop(sem, &op, 1);
        usleep(1000000);
   }
}
int main () {
   sem = semget(IPC_PRIVATE, 1, 0600 | IPC_CREAT);
   semctl(sem, 0, SETVAL, 1);
   ...
```

semop (sem_id, sem_ops, no)

- **sem_id** – semaphores set identifier
- **sem_ops** – array of semaphore operations (to be performed as **atomic** operation)
- **no** – number of operations

Get semaphore set (with only 1 semaphore)

Initial semaphore value

Extended functionality

- "Timed" wait on locks or queue
  - `pthread_mutex_timedlock(mutex, time)`
  - `pthread_cond_timedwait`
  - `pthread_rwlock_timedrdlock/timedwrlock`
  - `sem_timedwait`
  - will not wait more than specified on lock/queue
    - if that time elapses and lock is not obtained, error is returned

- Non-blocking "try" functions:
  - `pthread_mutex_trylock`
  - `pthread_rwlock_tryrdlock/trywrlock`
  - `pthread_spin_trylock`
  - `sem_trywait`, `semop` with `IPC_NOWAIT` flag
  - if locking can't be done – if it is already locked, will not block thread – instead will return error code

Programming problem: thread-safe functions

- Thread-safe, MT-safe (MultiThreading), reentrant ?
  - functions can be simultaneously (or even in parallel) called by different threads, and still produce valid results (same as if called only by single thread, sequentially)
- Some *library* or other functions may not be thread-safe (e.g. **`gethostbyname`**, **`rand`**)!
  - they use (internally) global variables (buffers, pointers…)
  - check function description (“man pages”)!
    - e.g. http://www.opengroup.org/onlinepubs/9699919799/functions/V2_chap02.html#tag_15_09
  - manually protect those “unsafe” function (e.g. with mutex)
- Build thread-safe functions
  - avoid global variables, or use them only in monitors