Threads: appendix
Processes vs. Threads

(a) Three processes each with one thread
(b) One process with three threads
Processes vs. Threads

<table>
<thead>
<tr>
<th>Per Process Items</th>
<th>Per Thread Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program counter</td>
</tr>
<tr>
<td>Global variables</td>
<td>Registers state</td>
</tr>
<tr>
<td>Open files</td>
<td>Stack</td>
</tr>
<tr>
<td>Child processes</td>
<td></td>
</tr>
<tr>
<td>Pending alarms</td>
<td></td>
</tr>
<tr>
<td>Signals and signal handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting information</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Each thread executes separately
- Threads in the same process share many resources
- No protection among threads!!
Thread Usage: Word Processor
Thread Usage: Web Server

Web server process

Dispatcher thread

Worker thread

Web page cache

Kernel

Network connection

User space

Kernel space

Copyright ©: University of Illinois CS 241 Staff
Summary: Creating Threads

- Initially, `main()` has a single thread
  - All other threads must be explicitly created
- `pthread_create()` ➔ new executable thread
  - Can be called any number of times from anywhere
- Maximum number of threads is implementation dependent

Question:
- After a thread has been created, how do you know when it will be scheduled to run by the operating system?
  - Answer: It is up to the operating system
    - Note: Good coding should not require knowledge of scheduling
void pthread_exit(void *value_ptr);

Common uses:

value_ptr is often a pointer to a malloc’d struct (memory must be free’d by joining thread)

Pass a pointer to heap not to the stack!!!
Thread Exit: pthread_exit or return?

• When a thread is done, it can return from its first function (the one used by pthread_create) or it can call pthread_exit.

• An implicit call to pthread_exit() is made when a thread other than main() returns from the start routine that was used to create it. The function’s return value shall serve as the thread’s exit status.
pthreads Attributes

Attributes
- Data structure `pthread_attr_t`
- Set of choices for a thread
- Passed in thread creation routine

Choices
- Scheduling options (more later on scheduling)
- Detached state
  - Detached
    - Main thread does not wait for created threads to terminate
  - Joinable
    - Main thread waits for created thread to terminate
    - Useful if created thread returns a value
pthreads Attributes

- Initialize an attributes structure to the default values
  - `int pthread_attr_init (pthread_attr_t* attr);`
- Set the detached state value in an attributes structure
  - `int pthread_attr_setdetachstate (pthread_attr_t* attr, int value);`
  - Value
    - `PTHREAD_CREATE_DETACHED`
    - `PTHREAD_CREATE_JOINABLE`
Waiting for Threads: `pthread_join()`

```c
int pthread_join(pthread_t thread, void** retval);
```

- **Note**
  - You cannot call `pthread_join()` on a detached thread,
  - Detaching means you are NOT interested in knowing about the thread’s exit

- Set `pthread_attr` to joinable before calling `pthread_create()`
  - This is the default option!

```c
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,
  PTHREAD_CREATE_JOINABLE);
```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void *snow(void *data) {
    printf("Main thread said:%s\n", (char *) data);
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    pthread_t mythread;
    int result;
    char *data = "Let it snow."
    result = pthread_create(&mythread, NULL, snow, (void *) data);
    printf("pthread_create() returned %d\n", result);
    if(result)
        exit(1);
    pthread_exit(NULL);
}

Example: `pthread_create()`
Example 1: process vs. thread

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

int x = 1;
void *func(void *p){
    x = x + 10;
    printf("child's x is %d\n", x);
    return NULL;
}

int main(int argc, char** argv) {
    if (fork() == 0)
        func(NULL);
    else {
        wait(NULL);
        printf("parent's x is %d\n", x);
    }
}
```
#include <stdio.h>
#include <pthread.h>

int x = 1;

void *func(void *p){
    x = x + 10;
    printf("func thread's x is %d\n", x);
    pthread_exit(NULL);
}

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

    pthread_create(&tid, NULL, func, NULL);
    pthread_join(tid, NULL);
    printf("main thread's x is %d\n", x);
}

Copyright ©: University of Illinois CS 241 Staff
Returning data through `pthread_join()`

```c
void *thread(void *vargp) {
    int *value = (int *)malloc(sizeof(int));
    *value = 84;
    pthread_exit(value);
}

int main() {
    int i; pthread_t tid; void *vptr_return;

    pthread_create(&tid, NULL, thread, NULL);
    pthread_join(tid, &vptr_return);
    i = *((int *)vptr_return);
    free(vptr_return);
    printf("%d\n",i);
}
```
Thread Argument Passing

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 8

char *messages[NUM_THREADS];

void *PrintHello(void *threadid) {
    int *id_ptr, taskid;

    sleep(1);
    id_ptr = (int *) threadid;
    taskid = *id_ptr;
    printf("Thread %d: %s\n", taskid, messages[taskid]);
    pthread_exit(NULL);
}
```
```c
int main(int argc, char *argv[]) {

    pthread_t threads[NUM_THREADS];
    int *taskids[NUM_THREADS];
    int rc, t;

    messages[0] = "English: Hello World!";
    messages[1] = "French: Bonjour, le monde!";
    messages[3] = "Klingon: Nuq neH!";
    messages[4] = "German: Guten Tag, Welt!";
    messages[5] = "Russian: Zdravstvytye, mir!";
    messages[7] = "Italian: Ciao Mondo!";
}```
Thread Argument Passing

```c
for(t=0; t<NUM_THREADS; t++) {
    taskids[t] = (int *) malloc(sizeof(int));
    *taskids[t] = t;
    printf("Creating thread %d\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello,
                       (void *) taskids[t]);
    if (rc) {
        printf("ERR; pthread_create() ret = %d\n", rc);
        exit(-1);
    }
}
pthread_exit(NULL);
```
Thread Argument Passing

```c
for(t=0;t<NUM_THREADS;t++) {
    taskids[t] = (int *) malloc(sizeof(int));
    *taskids[t] = t;
    printf("Creating thread %d\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello,
                       (void *) taskids[t]);
    if (rc) {
        printf("ERR; pthread_create() ret = %d\n", rc);
        exit(-1);
    }
}
pthread_exit(NULL);
```

Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
Creating thread 5
Creating thread 6
Creating thread 7

Thread 0: English: Hello World!
Thread 1: French: Bonjour, le monde!
Thread 2: Spanish: Hola el mundo!
Thread 3: Klingon: Nuq neH!
Thread 4: German: Guten Tag, Welt!
Thread 5: Russian: Zdravstvytye, mir!
Thread 6: Japanese: Sekai e konnichiwa!
Thread 7: Italian: Ciao Mondo!
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 8

char *messages[NUM_THREADS];

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

struct thread_data thread_data_array[NUM_THREADS];
Passing Complex Arguments

```c
void *PrintHello(void *threadarg) {
    int taskid, sum;
    char *hello_msg;
    struct thread_data *my_data;

    sleep(1);
    my_data = (struct thread_data *) threadarg;
    taskid = my_data->thread_id;
    sum = my_data->sum;
    hello_msg = my_data->message;
    printf("Thread %d: %s  Sum=%d\n", taskid, hello_msg, sum);
    pthread_exit(NULL);
}
```
int main(int argc, char *argv[]) {

    pthread_t threads[NUM_THREADS];
    int rc, t, sum;

    sum=0;
    messages[0] = "English: Hello World!";
    messages[1] = "French: Bonjour, le monde!";
    messages[3] = "Klingon: Nuq neH!";
    messages[4] = "German: Guten Tag, Welt!";
    messages[5] = "Russian: Zdravstvuyte, mir!";
    messages[7] = "Italian: Ciao Mondo!";
}
Passing Complex Arguments

```c
for(t=0; t<NUM_THREADS; t++) {
    sum = sum + t;
    thread_data_array[t].thread_id = t;
    thread_data_array[t].sum = sum;
    thread_data_array[t].message = messages[t];
    printf("Creating thread %d\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello,
                        (void *) &thread_data_array[t]);
    if (rc) {
        printf("ERR; pthread_create() ret = %d\n", rc);
        exit(-1);
    }
}
pthread_exit(NULL);
```
for(t=0; t<NUM_THREADS; t++) {
    sum = sum + t;
    thread_data_array[t].thread_id = t;
    thread_data_array[t].sum = sum;
    rc = pthread_create(&threads[t], NULL, PrintHello,
                        (void *) taskids[t]);
    if (rc) {
        printf("ERR; pthread_create() ret = %d\n", rc);
        exit(-1);
    }
}

pthread_exit(NULL);
Passing Complex Arguments

```c
for(t=0;t<NUM_THREADS;t++) {
    sum = sum + t;
    thread_data_array[t].thread_id = t;
    thread_data_array[t].sum = sum;
    thread_data_array[t].message = messages[t];
    printf("Creating thread %d
", t);
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &thread_data_array[t]);
    if (rc) {
        printf("ERR; pthread_create() ret = %d
", rc);
        exit(-1);
    }
    printf(" Thread %d: English: Hello World!  Sum=%d
", t, sum);
    printf(" Thread %d: French: Bonjour, le monde!  Sum=%d
", t, sum);
    printf(" Thread %d: Spanish: Hola el mundo!  Sum=%d
", t, sum);
    printf(" Thread %d: Klingon: Nuq neH!  Sum=%d
", t, sum);
    printf(" Thread %d: German: Guten Tag, Welt!  Sum=%d
", t, sum);
    printf(" Thread %d: Russian: Zdravstvytye, mir!  Sum=%d
", t, sum);
    printf(" Thread %d: Japan: Sekai e konnichiwa!  Sum=%d
", t, sum);
    printf(" Thread %d: Italian: Ciao Mondo!  Sum=%d
", t, sum);
}
pthread_exit(NULL);
```
Incorrect Argument Passing

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 8

void *PrintHello(void *threadid)
{
    int *id_ptr, taskid;
    sleep(1);
    id_ptr = (int *) threadid;
    taskid = *id_ptr;
    printf("Hello from thread %d\n", taskid);
    pthread_exit(NULL);
}
```
int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc, t;

    for(t=0;t<NUM_THREADS;t++) {
        printf("Creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello,
                            (void *) &t);
        if (rc) {
            printf("ERR; pthread_create() ret = %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
Incorrectly returning data through `pthread_join()`

```c
void *thread(void *vargp) {
    exit(42);
}

int main() {
    int i;
    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);
    pthread_join(tid, (void **)&i);
    printf("%d\n",i);
}
```

What will happen here?
Incorrectly returning data through \texttt{pthread_join()}

```c
void *thread(void *vargp) {
    pthread_detach(pthread_self());
    pthread_exit((void*)42);
}

int main() {
    int i = 0;
    pthread_t tid;

    pthread_create(&tid, NULL, thread, NULL);
    pthread_join(tid, (void**)&i);
    printf("%d\n",i);
}
```

What will happen here?
Common Ways to Structure Multi-threaded Code

- Manager/worker
  - Single thread (manager) assigns work to other threads (workers)
  - Manager handles all input and parcels out work
Manager/Worker Model

Manager:
create N workers
forever {
    get a request
    pick free worker
}

Worker:
forever {
    wait for request
    perform task
}
Common Ways to Structure Multi-threaded Code

- Pipeline
  - Task is broken into a series of sub-tasks
  - Each sub-task is handled by a different thread
Pipeline Model

Manager:
create N stages
forever {
    get a request
    pick 1st stage
}

Stage N:
forever {
    wait for request
    perform task
    pick stage n+1
}
on GNU/Linux, threads are implemented as processes. Whenever you call `pthread_create` to create a new thread, Linux creates a new “light” process that runs that thread.

Each thread maps to a kernel scheduling entity. Scheduler handles pthreads as regular processes (you can assign a scheduling priority to each thread!)

`pthread_t` identifier is MEANINGFUL only in the process that created it and is not visible outside. So for instance, you cannot send a `pthread_kill` to a thread of another process.
More details for Linux:

- `pthread_self()` will get you an identifier that is unique across your program, but not across your system. Although thread is a system object, the system is unaware of the identifier POSIX library allocated for the thread. On the contrary, Linux identifies threads with PID like number called TID: these numbers are system-wide.

- Each thread in a process has a different thread ID and shares the same Process ID. If you are working with pthread library functions, these functions don’t use these TIDs because these are kernel/OS level \textit{non-POSIX} thread IDs.

- In a single-threaded process, the Thread ID is equal to the Process ID (PID, as returned by `getpid(2)`). In a multithreaded process, all threads have the same PID, but each one has a unique TID.
More details for Linux:

- Do you want to see your threads listed when invoking `ps`?
  - Try: `ps -T`

<table>
<thead>
<tr>
<th>PID</th>
<th>SPID</th>
<th>TTY</th>
<th>TIME</th>
<th>CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>4932</td>
<td>4932</td>
<td>pts/1</td>
<td>00:00:00</td>
<td>xterm</td>
</tr>
<tr>
<td>4935</td>
<td>4935</td>
<td>pts/3</td>
<td>00:00:00</td>
<td>bash</td>
</tr>
<tr>
<td>4937</td>
<td>4937</td>
<td>pts/1</td>
<td>00:00:00</td>
<td>a.out</td>
</tr>
<tr>
<td>4937</td>
<td>4938</td>
<td>pts/1</td>
<td>00:09:01</td>
<td>a.out</td>
</tr>
<tr>
<td>4967</td>
<td>4967</td>
<td>pts/3</td>
<td>00:00:00</td>
<td>ps</td>
</tr>
</tbody>
</table>