

Threads: use & systems view

CS 241

February 22, 2012

Copyright © University of Illinois CS 241 Staff

Today

MP4 released

- Multithreaded merge sort

Terminating threads

- Finishing up from last time

Whose memory is it, anyway?

- Passing arguments
- Thread safety

Systems view of threads

- User space vs. kernel

Terminating threads

Terminating Threads: `pthread_exit()`

```
int pthread_exit(void * retval);
```

Terminate the calling thread

Makes the value `retval` available to any successful join with the terminating thread

Returns

- `pthread_exit()` cannot return to its caller

Parameters

- `retval`: Pointer to data returned to joining thread

Note

- If `main()` exits before its threads via `pthread_exit()`, the other threads continue. Otherwise, they will be terminated when `main()` ends.

Termination example


```
#include <pthread.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid) {
    printf("\n%d: Hello World!\n", threadid);
    pthread_exit(NULL);
}
```

Termination example

```
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("ERROR; pthread_create() return code is %d\n",
                   rc);
            exit(-1);
        }
    }
}
pthread_exit(NULL);
```



Will all threads get a chance to execute?

Termination example

```
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("ERROR; pthread_create() return code is %d\n",
                  rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

Will all threads get a chance to execute before the parent exits?

```
for(t=0;t < NUM_THREADS;t++) {
    pthread_join(threads[t], NULL);
    printf("Joined thread %d\n",t);
}
```

Thread Lifetime

A thread exists until...

- It returns from the function or calls `pthread_exit()`
- The whole process terminates
- The machine catches fire

So, your process terminates when...

Any thread calls `exit()`;

The main thread returns

- ```
main() {
 pthread_create();
 return 0;
}
```

Segmentation fault

- ```
*(char*)0 = 0;
```

There are no more threads left to run

**Whose memory is it,
anyway?**

Example: argument passing

```
#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);
}
```

Example: argument passing

```
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);
}
```

Does this code work?

Example: argument passing

```
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);
}
```

The loop that creates threads modifies the contents of the address passed as an argument, possibly before the created threads can access it.

What is the possible output?

Candidate “contracts” for thread args

main owns the memory

- main creates memory (e.g., integer arguments from last example)
- Threads can use it briefly to get their argument
- main can later modify & is responsible for freeing

child thread owns the memory


- main malloc’s memory
- main transfers “ownership” of argument memory to thread at startup
- thread can read/write/free, main can’t touch it (until thread is done)

nobody owns the memory

- global variable; nothing to free()
- once created by main(), no one writes to the memory

Candidate “contracts” for thread args

Xmain owns the memory

- main creates memory (e.g., integer arguments from last example)
 - Threads can use it briefly to get their argument
 - main can later modify & is responsible for freeing
- 
- Conflict

child thread owns the memory

- main malloc's memory
- main transfers “ownership” of argument memory to thread at startup
- thread can read/write/free, main can't touch it (until thread is done)

nobody owns the memory

- global variable; nothing to free()
- once created by main(), no one writes to the memory

Better argument passing

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


Better argument passing

```
#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);
    free(threadid);
    pthread_exit(NULL);
}
```

PrintHello thread owns the memory; so PrintHello is responsible for freeing memory

How about this?

```
pthread_t threads[NUM_THREADS];

void make_thread(int arg) {
    rc = pthread_create(&threads[arg], NULL, PrintHello,
                       (void *) &arg);
    if (rc) {
        printf("ERR; pthread_create() ret = %d\n", rc);
        exit(-1);
    }
}

int main() {
    for(t=0;t<NUM_THREADS;t++) {
        make_thread(t);
    }
    pthread_exit(NULL);
}
```

How about this?

```
pthread_t threads[NUM_THREADS];

void make_thread(int arg) {
    rc = pthread_create(&threads[arg], NULL, PrintHello,
                       (void *) &arg);
    if (rc) {
        printf("ERR; pthread_create() ret = %d\n", rc);
        exit(-1);
    }
} ←
```

Violates “thread owns” rule:
arg is destroyed here, while thread
is still running

```
int main() {
    for(t=0;t<NUM_THREADS;t++) {
        make_thread(t);
    }
    pthread_exit(NULL);
}
```

Example: pthread error handling

Recall **errno**: global error code variable

- Problem: which of the many threads “owns” **errno**?

Compared to normal TCP system calls, pthreads functions...

- Similarity
 - Returns 0 on success
- Differences
 - Returns error code on failure
 - Does not set **errno**
- What about **errno**?
 - Each thread has its own
 - Define **_REENTRANT** (**-D_REENTRANT** switch to compiler) when using pthreads

Pitfalls

Global variables

- Problem: No protection between threads
- Solutions:
 - Disallow all global variables
 - Introduce new thread-specific global variables

Are my libraries thread-safe?

- May use local variables
- May not be designed to be interrupted
 - Create wrappers

Unsafe Library Calls

```
#include <string.h>
```

```
char *token;
```

```
char *line = "LINE TO BE SEPARATED";
```

```
char *search = " ";
```

```
/* Token will point to "LINE". */
```

```
token = strtok(line, search);
```

```
/* Token will point to "TO". */
```

```
token = strtok(NULL, search);
```

```
#include <string.h>
```

```
char *token;
```

```
char *line = "LINE TO BE SEPARATED";
```

```
char *search = " ";
```

```
/* Token will point to "LINE". */
```

```
token = strtok_r(line, search);
```

```
/* Token will point to "TO". */
```

```
token = strtok_r(NULL, search);
```

Thread-safe Library Calls

```
#include <string.h>
```

```
char *token;
```

```
char *line = "LINE TO BE  
SEPARATED";
```

```
char *search = " ";
```

```
char *state;
```

```
/* Token will point to "LINE". */
```

```
token = strtok_r(line, search,  
&state);
```

```
#include <string.h>
```

```
char *token;
```

```
char *line = "LINE TO BE  
SEPARATED";
```

```
char *search = " ";
```

```
char *state;
```

```
/* Token will point to "LINE". */
```

```
token = strtok_r(line, search,  
&state);
```

System & library functions that are not required to be thread-safe

asctime	dirname	getenv	getpwent	lgamma	readdir
basename	dlderror	getgrent	getpwnam	lgammaf	setenv
catgets	drand48	getgrgid	getpwuid	lgammal	setgrent
crypt	ecvt	getgrnam	getservbyname	localeconv	setkey
ctime	encrypt	gethostbyaddr	getservbyport	localtime	setpwent
dbm_clearerr	endgrent	gethostbyname	getservent	lrand48	setutxent
dbm_close	endpwent	gethostent	getutxent	mrnd48	strerror
dbm_delete	endutxent	getlogin	getutxid	nftw	strtok
dbm_error	fcvt	getnetbyaddr	getutxline	nl_langinfo	ttyname
dbm_fetch	ftw	getnetbyname	gmtime	ptsname	unsetenv
dbm_firstkey	gcvt	getnetent	hcreate	putc_unlocked	wcstombs
dbm_nextkey	getc_unlocked	getopt	hdestroy	putchar_unlocked	wctomb
dbm_open	getchar_unlocked	getprotobynumber	inet_ntoa	pututxline	
dbm_store	getdate	getprotoent	l64a	rand	

Key take-away points

Easiest way to coordinate between threads:

- Be clear which thread “owns” a piece of memory at any time
- Others must not write to it, or destroy it via `free()` or via returning from a function
- Not everything falls into this category

Make sure your library calls are thread-safe

All the above only works if **one** thread needs the memory

- ...so we can have an “owner”
- General-purpose coordination between threads: next week

The Operating System's view of threads

Thread Packages

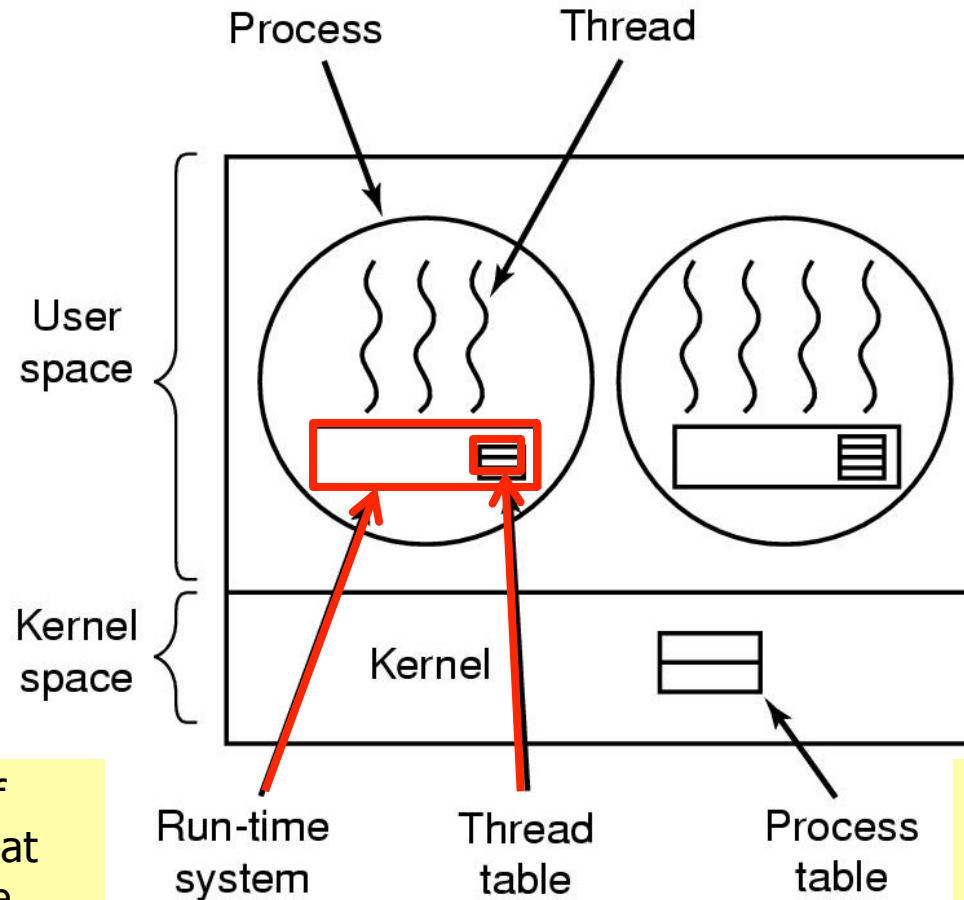
Kernel thread packages

- Implemented and supported at kernel level

User-level thread packages

- Implemented at user level
- Kernel perspective: everything is a single-threaded process

Threads in User Space (Old Linux)



Collection of procedures that manages the threads

Keep track of threads in process (analogous to kernel process table)

User-level Threads

Advantages

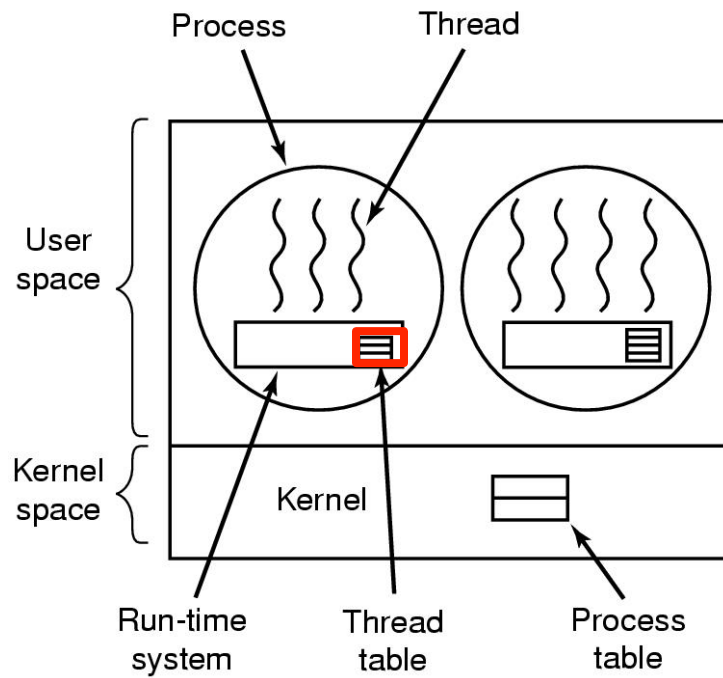
- Fast Context Switching: keeps the OS out of it!
 - User level thread libraries do not require system calls
 - No call to OS and no interrupts to kernel
 - `thread_yield`
 - Save the thread information in the thread table
 - Call the thread scheduler to pick another thread to run
 - Saving local thread state scheduling are local procedures
 - No trap to kernel, low context switch overhead, no memory switch
- Customized Scheduling (at user level)

User-level Threads

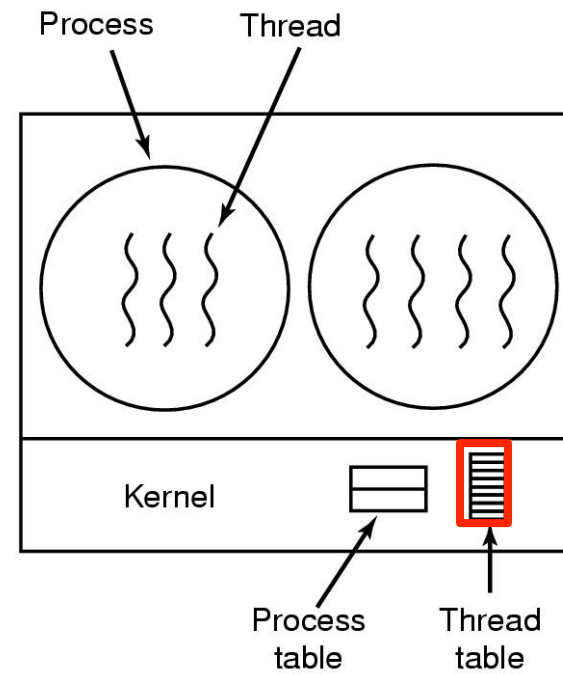
Disadvantages

- What happens if one thread makes a blocking I/O call?
 - Change the system to be non-blocking
 - Always check to see if a system call will block
- What happens if one thread never yields?
 - Introduce clocked interrupts
- Multi-threaded programs frequently make system calls
 - Causes a trap into the kernel anyway!

User vs. Kernel Threads



User-level Threads



Kernel-level Threads

Kernel-level Threads

Advantages

- Kernel schedules threads in addition to processes
- Multiple threads of a process can run simultaneously
 - Now what happens if one thread blocks on I/O?
 - Kernel-level threads can make blocking I/O calls without blocking other threads of same process
- Good for multicore architectures

Kernel-level Threads

Disadvantages

- Overhead in the kernel... extra data structures, scheduling, etc.
- Thread creation is expensive
 - Have a pool of waiting threads
- What happens when a multi-threaded process calls `fork()`?
- Which thread should receive a signal?

Trade-offs?

Kernel thread packages

- Each thread can make blocking I/O calls
- Can run concurrently on multiple processors

Threads in User-level

- Fast context switch
- Customized scheduling
- No need for kernel support

Things to think about for the future

Who gets to go next when a thread blocks/yields?

- Scheduling!

What happens when multiple threads are sharing the same resource?

- Synchronization!