



Threads

Processes versus Threads

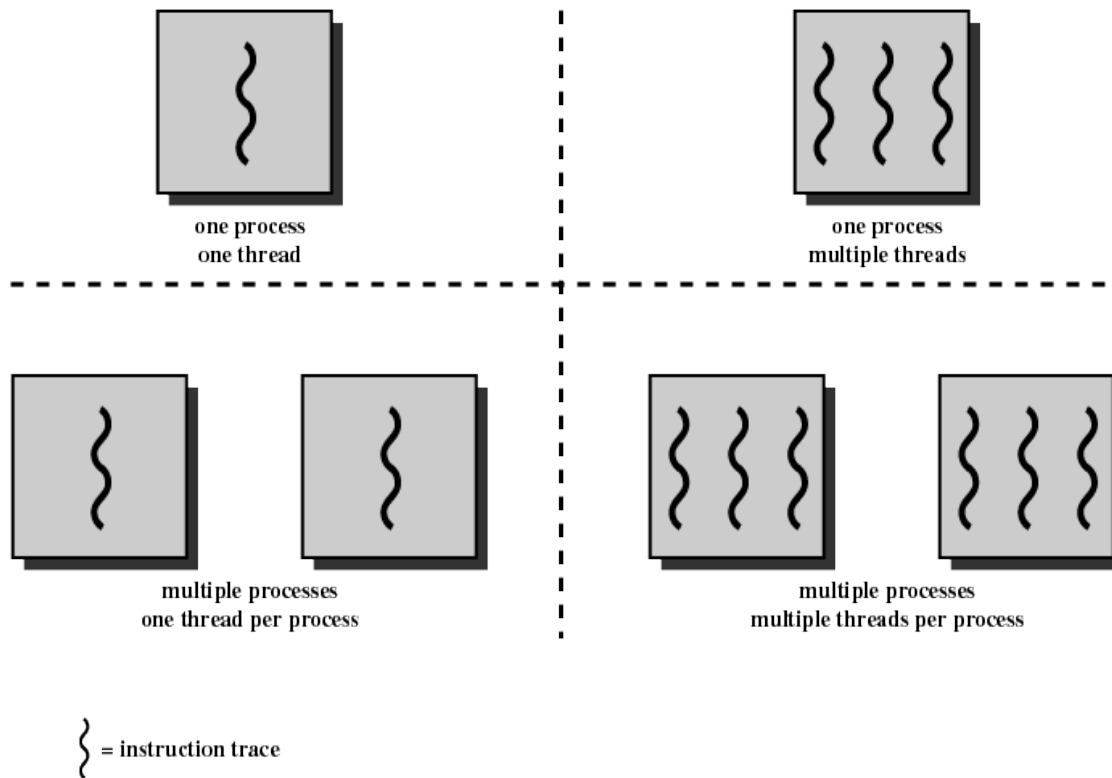


Figure 4.1 Threads and Processes [ANDE97]

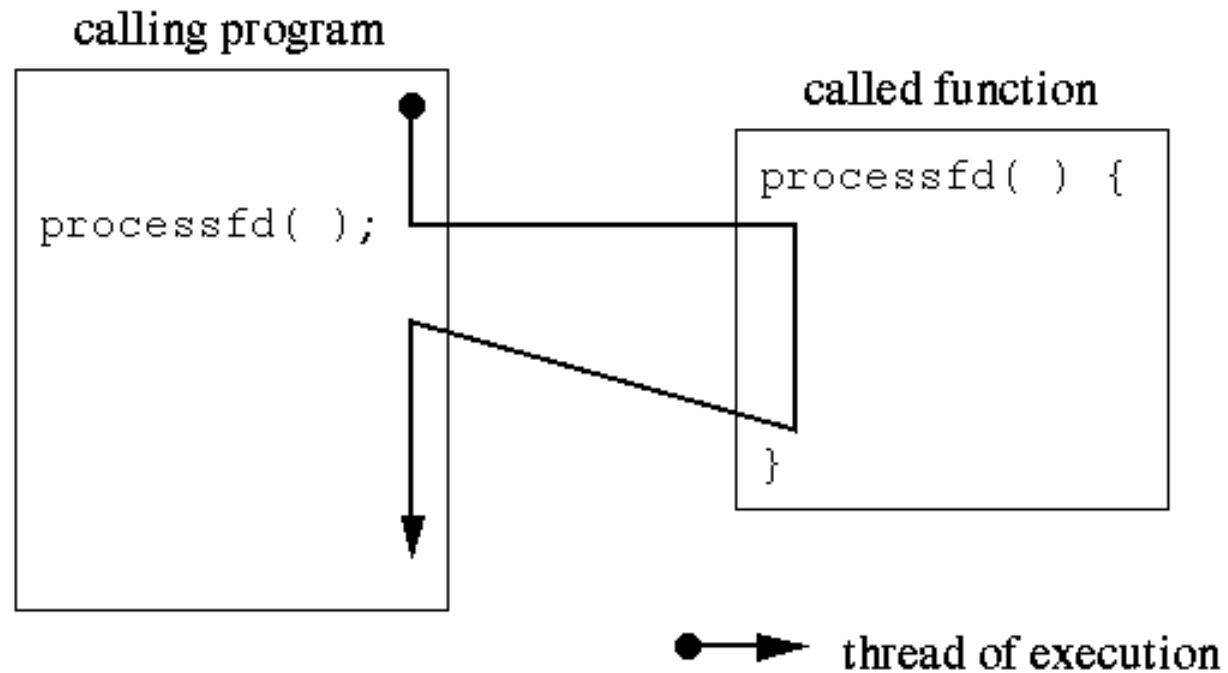


Creating a Thread

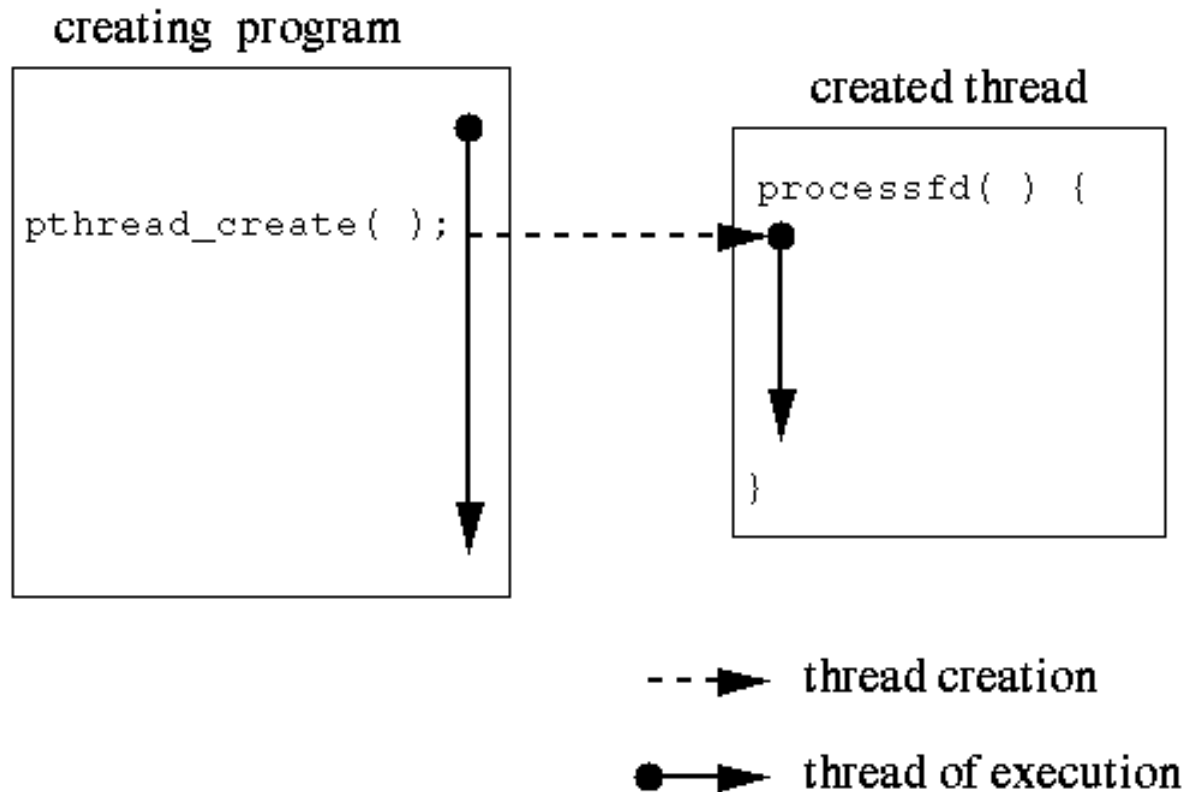
- When a new thread is created it runs concurrently with the creating process.
- When creating a thread you indicate which function the thread should execute.



Normal function call



Threaded function call





Threads vs. Processes

Per process items

Address space
Global variables
Open files
Child processes
Pending alarms
Signals and signal handlers
Accounting information

Per thread items

Program counter
Registers
Stack
State

- Each thread executes separately
- Threads in the same process share resources
- No protection among threads!!



Threads versus Processes

Property	Processes created with fork	Threads of a process	Ordinary function calls
variables	get copies of all variables	share global variables	share global variables
IDs	get new process IDs	share the same process ID but have unique thread ID	share the same process ID (and thread ID)
Communication	Must explicitly communicate, e.g.pipes or use small integer return value	May communicate with return value or shared variables if done carefully	May communicate with return value or shared variables (don't have to be careful)
Parallelism (one CPU)	Concurrent	Concurrent	Sequential
Parallelism (multiple CPUs)	May be executed simultaneously	Kernel threads may be executed simultaneously	Sequential



Pthread Operations

POSIX function	description
<code>pthread_create</code>	create a thread
<code>pthread_detach</code>	set thread to release resources
<code>pthread_equal</code>	test two thread IDs for equality
<code>pthread_exit</code>	exit a thread without exiting process
<code>pthread_kill</code>	send a signal to a thread
<code>pthread_join</code>	wait for a thread
<code>pthread_self</code>	find out own thread ID



Thread Packages

- Kernel thread packages
 - Implemented and supported at kernel level
- User-level thread packages
 - Implemented at user level

POSIX threads on GNU/Linux



- on GNU/Linux, threads are implemented as processes. Whenever you call `pthread_create` to create a new thread, Linux creates a new process that runs that thread.
- A posix thread is not the same as a process you would create with `fork`; in particular, it shares the same address space and resources as the original process rather than receiving copies.
- Each thread maps to a kernel scheduling entity.

POSIX threads on GNU/Linux



- How is a pthread type of process created on GNU/Linux?
- The Linux clone system call is a generalized form of fork and pthread_create that allows the caller to specify which resources are shared between the calling process and the newly created process.
- Clone system call should not ordinarily be used in programs. Use fork to create new processes or pthread_create to create threads. **WHY?**

POSIX threads on GNU/Linux

- How is a pthread type of process created on GNU/Linux?
- The Linux clone system call is a generalized form of fork and pthread_create that allows the caller to specify which resources are shared between the calling process and the newly created process.
- Clone system call should not ordinarily be used in programs. Use fork to create new processes or pthread_create to create threads. **WHY?**
➔ it is not part of POSIX standard!



pthread_t identifier

- 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.



Example Program

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

```
#include <stdio.h>
```

```
void *threadex(void *);
```

```
int main() {
```

```
    pthread_t tid; /* stores the new thread ID */
```

```
    pthread_create(&tid, NULL, threadex, NULL); /*create a new thread*/
```

```
    pthread_join(tid, NULL); /*main thread waits for new thread to terminate */
```

```
    return 0; /* main thread exits */
```

```
}
```

```
void *threadex(void *arg) /*thread routine*/ {
```

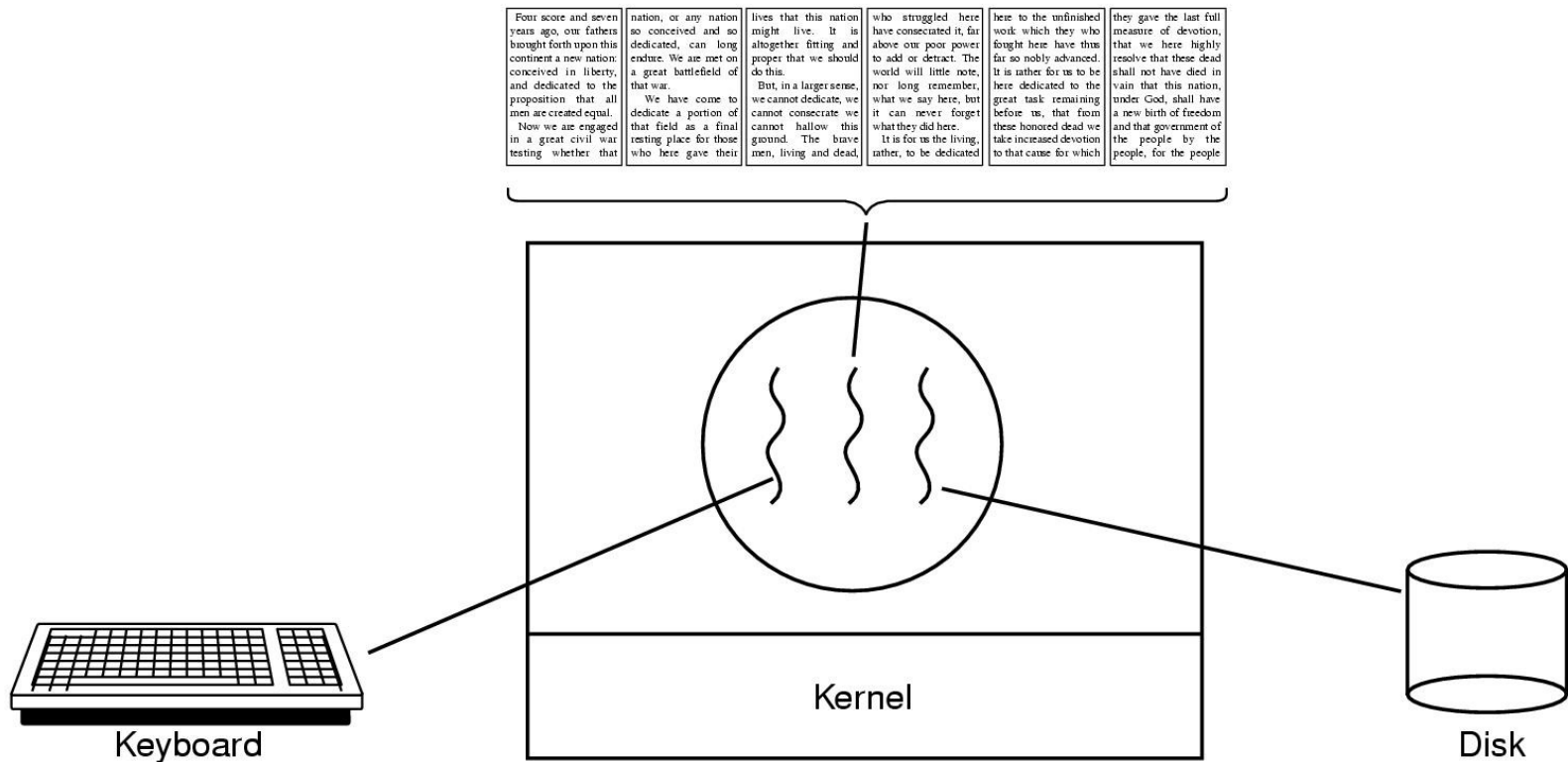
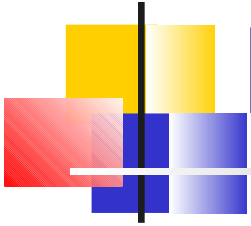
```
    int i;
```

```
    for (i=0; i<5; i++)
```

```
        fprintf(stderr, "Hello, world! \n ");
```

```
    return NULL; }
```

Thread Usage: Word Processor

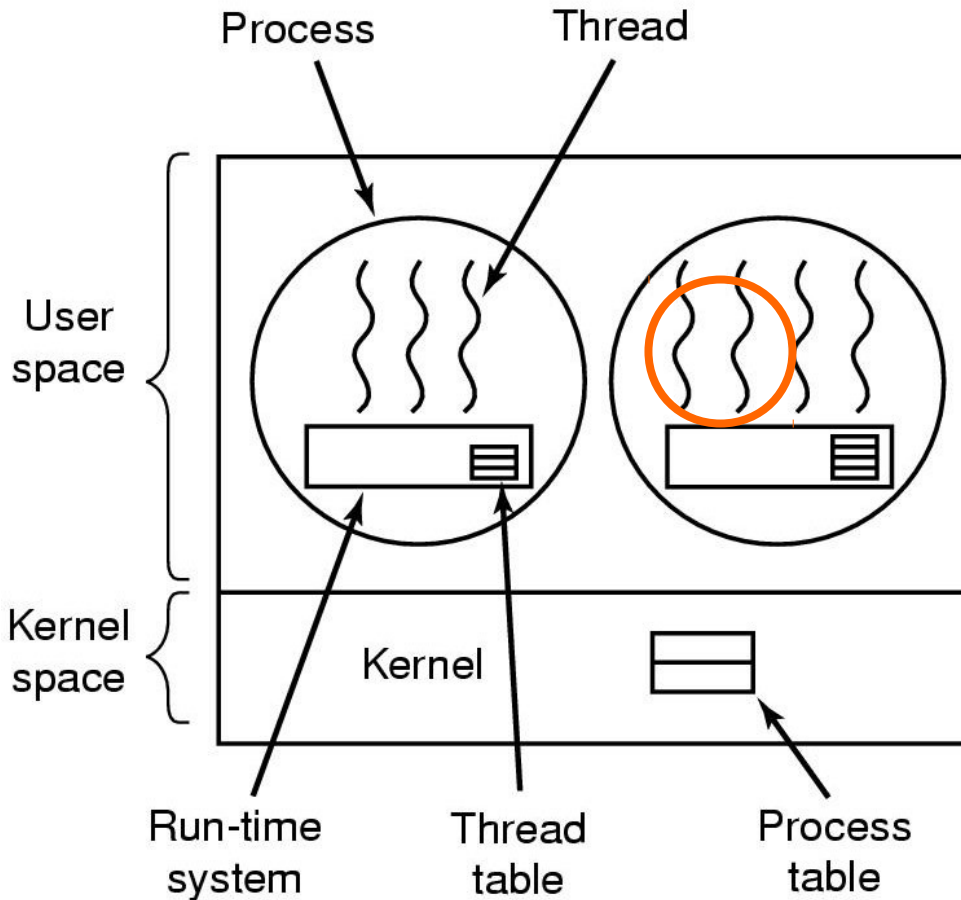


- What if it is single-threaded?

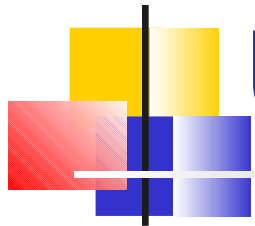
Things Suitable for threading

- Block for potentially long waits
- Use many CPU cycles
- Must respond to asynchronous events
- Are able to be performed in parallel with other tasks

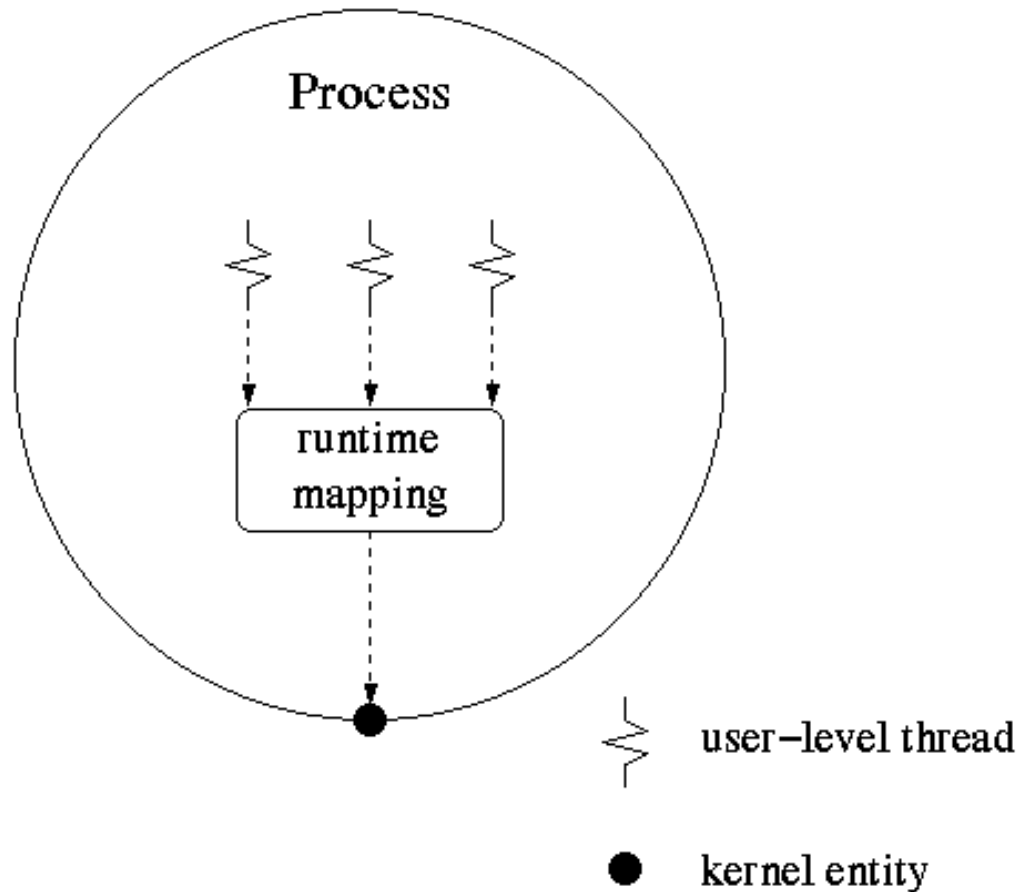
Implementing Threads in User Space (Old Linux)



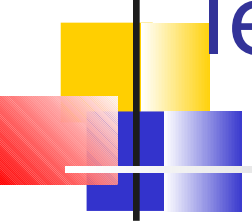
A user-level
threads
package



User-level Threads



Trade-offs between Kernel & user-level threads?

- 
-
- Kernel thread packages
 - Each thread can make blocking I/O calls without blocking the entire process
 - Can run concurrently on multiple processors
 - Threads in User-level
 - Fast context switch
 - Customized scheduling



Creating a thread with pthread

A thread is created with

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

The creating process (or thread) must provide a location for storage of the thread id.

The third parameter is just the name of the function for the thread to run.

The last parameter (void *arg) is the sole argument passed to created thread.

Example 1: Thread Creation



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

void *PrintHello(void *threadid) {
    int tid;
    tid = (int)threadid;
    printf("Hello World! It's me, thread #%%d!\n", tid);
    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("In main: creating thread %%d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc) {
            printf("ERROR code is %%d\n", rc);
            exit(1); }
    }
    pthread_exit(NULL);
}
```



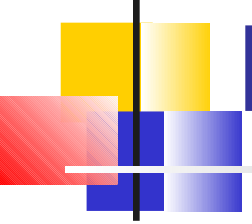
Example 2: Passing Parameters to a Thread

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS    5

void *PrintHello(void *ptr)
{
    char *filename;
    int j;
    filename = (char *) ptr;
    while (1) {
        printf("Hello World! It's me, thread %s!\n", filename);
        for (j=1; j++; j<500);
    }
    pthread_exit(NULL);
}
```

Example 2: Passing Parameters to a Thread



```
int main (int argc, char *argv[])
{
    pthread_t thread[100];
    int err_code, i=0;
    char *filename;

    printf ("Enter thread name at any time to create thread\n");

    while (i <= 99) {
        filename = (char *) malloc (80*sizeof(char));
        scanf ("%s", filename);

        printf("In main: creating thread %d\n", i);
        err_code = pthread_create(&thread[i], NULL, PrintHello, (void *)filename);
        if (err_code){
            printf("ERROR code is %d\n", err_code);
            exit(1);
        }
        i++;
    }
    pthread_exit(NULL);
}
```



Example 3: Files

```
void *PrintHello(void *ptr)
{
    FILE *file;
    char *filename;
    char textline[100];
    int j;

    filename = (char *) ptr;
    printf("Hello World! Opening %s!\n", filename);
    file = fopen(filename, "r");
    if (file != NULL) {
        while(fscanf(file, "%s\n", textline) != EOF) printf ("%s\n", textline);
        fclose(file);
    }
}
```




Exiting and Cancellation

- Question:
 - If a thread calls `exit()`, what about other threads in the same process?
- When does a process terminate?



Exiting and Cancellation

- Question:
 - If a thread calls `exit()`, what about other threads in the same process?
- A process can terminate when:
 - it calls `exit` directly
 - one of its threads calls `exit`
 - it returns from `main()`
 - it receives a termination signal
- In any of these cases, all threads of the process terminate.



Thread Exit

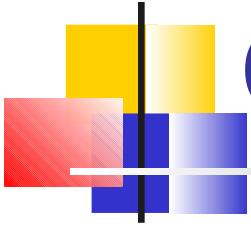
When a thread is done, it can return from its first function (the one used by `pthread_create`) or it can call `pthread_exit`

```
void pthread_exit(void *value_ptr);
```

Common uses:

Return value is often a pointer to the original struct or a malloc'd struct (memory must be free'd by joining thread)

Use the heap not the stack!!!



Cancel that thread!

- One thread can request that another exit with `pthread_cancel`
- `int pthread_cancel(pthread_t thread);`
- The `pthread_cancel` returns after making the request.



Thread Attributes

- Create an attribute object (initialize it with default properties)
- Modify the properties of the attribute object
- Create a thread using the attribute object
- The object can be changed or reused without affecting the thread
- The attribute object affects the thread only at the time of thread creation



Example: Create a detached thread

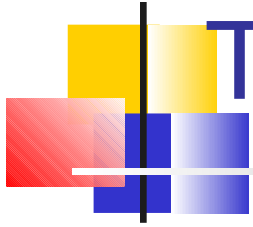
```
int e, fd;
pthread_attr_t tattr;
pthread_t tid;

if(e = pthread_attr_init(&tattr))
    fprintf(stderr, "Failed to create attribute object: %s\n", strerror(e));

else if(e = pthread_attr_setdetachstate(&tattr,
    PTHREAD_CREATE_DETACHED))
    fprintf(stderr, "Failed to set attribute state to detached: %s\n",
    strerror(e));

else if(e = pthread_create(&tid, &tattr, data, &fd))
    fprintf(stderr, "Failed to create thread: %s\n", strerror(e));
```

Settable Properties of Thread Attributes



property	function
attribute objects	pthread_attr_destroy
	pthread_attr_init
detach state	pthread_attr_getdetachstate
	pthread_attr_setdetachstate
stack	pthread_attr_getguardsize
	pthread_attr_setguardsize
	pthread_attr_getstack
	pthread_attr_setstack
scheduling	pthread_attr_getinheritsched
	pthread_attr_setinheritsched
	pthread_attr_getschedparam
	pthread_attr_setschedparam
	pthread_attr_getschedpolicy
	pthread_attr_setschedpolicy
	pthread_attr_getscope
	pthread_attr_setscope

Zombies, Thread Detach & Join

- Call **pthread_join()** or **pthread_detach()** for every thread that is created joinable
 - so that the system can reclaim all resources associated with the thread
- Failure to join or to detach threads
→ memory and other resource leaks until the process ends



Detaching a Thread

- `int pthread_detach(pthread_t threadid);`
- Indicate that system resources for the specified thread should be reclaimed when the thread ends
 - If the thread is already ended, resources are reclaimed immediately
 - This routine does not cause the thread to end
- A detached thread's thread ID is undetermined
- Threads are detached
 - after a `pthread_detach()` call
 - after a `pthread_join()` call
 - if a thread terminates and the `PTHREAD_CREATE_DETACHED` attribute was set on creation

How to make a Thread Detached

```
void *processfd(void *arg);
```

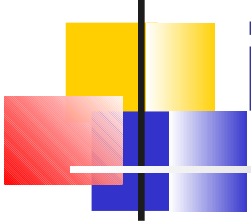
```
int error;
```

```
int fd
```

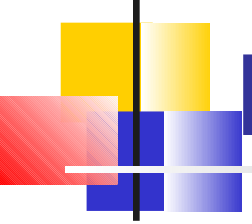
```
pthread_t tid;
```

```
if (error = pthread_create(&tid, NULL, processfd, &fd)) {  
    fprintf(stderr, "Failed to create thread: %s\n", strerror(error));  
}  
else if (error = pthread_detach(tid)){  
    fprintf(stderr, "Failed to detach thread: %s\n", strerror(error));  
}
```

How a thread can detach itself



```
void *runnger(void *arg) {  
    ...  
    if (!pthread_detach( pthread_self() ) )  
        return NULL;  
  
    ...  
}
```



“Waiting” on a Thread: `pthread_join()`

- `int pthread_join(pthread_t thread, void** retval);`
- `pthread_join()` is a *blocking call* on non-detached threads
- It indicates that the caller wishes to block until the thread being joined exits
- You cannot join on a detached thread, only non-detached threads (detaching means you are NOT interested in knowing about the threads exit)



Pthread_join

```
int error;  
int *exitcodep;  
pthread_t tid;  
  
if (error = pthread_join(tid, &exitcodep)){  
    fprintf(stderr, "Failed to join thread: %s\n", strerror(error));  
}  
else {  
    fprintf(stderr, "The exit code was %d\n", *exitcodep);  
}
```



Thread scheduling

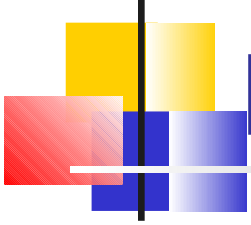
```
int pthread_attr_getscope(const pthread_attr_t *restrict attr,  
    int *restrict contentscope);
```

```
int pthread_attr_setscope(pthread_attr_t *attr, int contentscope);
```

The contentscope can be PTHREAD_SCOPE_PROCESS or PTHREAD_SCOPE_SYSTEM.

The scope determines whether the thread competes with other threads of the process or with other processes in the system.

Create a thread that contends with other processes



```
pthread_attr_init(&tattr)
```

```
pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM))
```

```
thread_create(&tid, &tattr, myfunction, &myptr))
```