

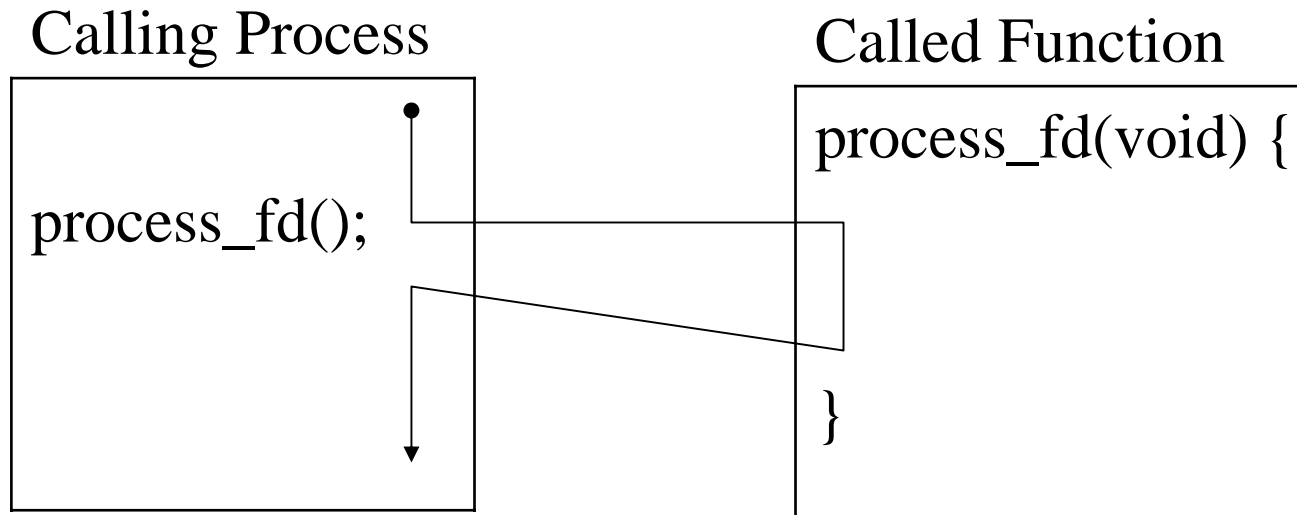
Threads

- Threads are lightweight processes
- In a context switch, they change the contents of the CPU registers but do not change memory
- Threads can simplify the programming of problems such as monitoring inputs from multiple file descriptors
- They also provide a capability to overlap I/O with processing
- Typical thread packages contain a runtime system to manage threads in a transparent way
- A thread package contains calls for thread creation and destruction, mutual exclusion, and condition variables
- `POSIX.THR` and Sun Solaris 2 standard libraries have such calls

Methods to Monitor Multiple File Descriptors

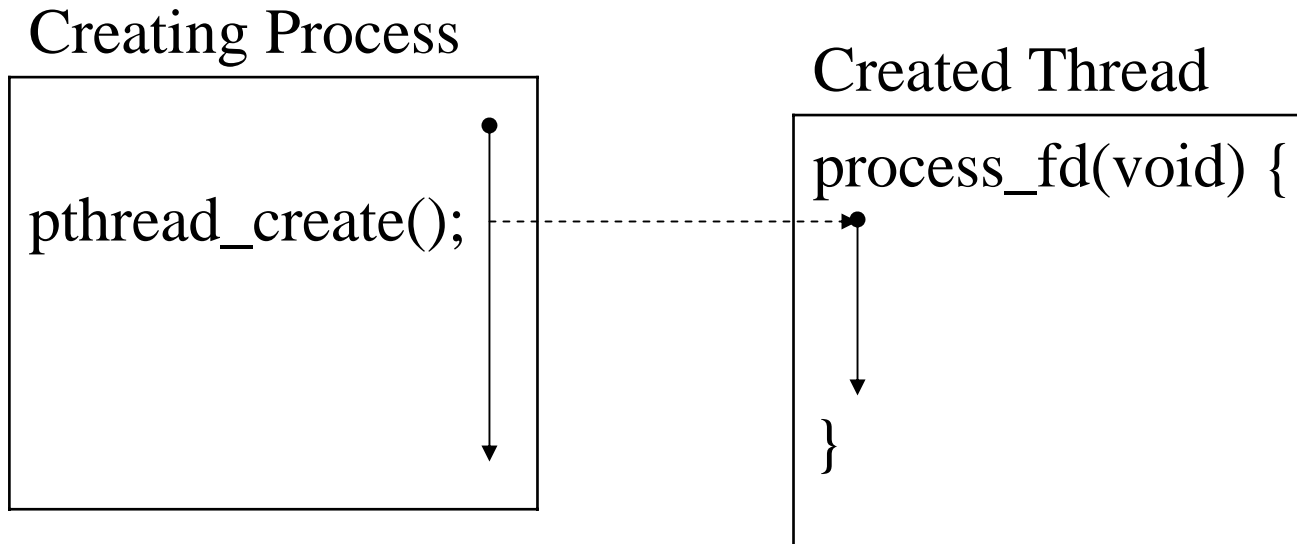
- Have a separate process monitor each file descriptor
- Use the select command
- Use the poll command
- Use POSIX asynchronous I/O
- Create a thread to monitor each file descriptor

Without Threads



• → Thread of execution

With Threads



-----> Thread Creation

•-----> Thread of execution

processfd.c

```
#include <stdio.h>
#include "restart.h"
#define BUFSIZE 1024
void docommand(char *cmd, int cmdsize);
void *processfd(void *arg) { /* process commands read from file
    descriptor */
    char buf[BUFSIZE];
    int fd;
    ssize_t nbytes;
    fd = *((int *)(arg));
    for ( ; ; ) { if ((nbytes = r_read(fd, buf, BUFSIZE)) <= 0)
        break; docommand(buf, nbytes); }
    return NULL; }
```

processid.c Analysis

- Processid.c monitors only one file descriptor for an input
- The input is a command for execution

monitorfd.c - top

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void *processfd(void *arg);
void monitorfd(int fd[], int numfds) {    /* create threads to monitor fds */
    int error, i;
    pthread_t *tid;
    if ((tid = (pthread_t *)calloc(numfds, sizeof(pthread_t))) == NULL) {
        perror("Failed to allocate space for thread IDs");
        return; }
}
```

monitorfd.c

```
for (i = 0; i < numfds; i++) /* create a thread for each file descriptor */
    if (error = pthread_create(tid + i, NULL, processfd, (fd + i))) {
        fprintf(stderr, "Failed to create thread %d: %s\n",
                i, strerror(error));
        tid[i] = pthread_self();
    }
for (i = 0; i < numfds; i++) {
    if (pthread_equal(pthread_self(), tid[i]))
        continue;
    if (error = pthread_join(tid[i], NULL))
        fprintf(stderr, "Failed to join thread %d: %s\n", i, strerror(error));
}
free(tid);
return;
}
```

POSIX vs Solaris 2 (1)

- Most thread functions return 0 if successful and nonzero error code if unsuccessful
- *pthread_create (thr_create)* – Creates a thread
- *pthread_exit (thr_exit)* – Causes the calling thread to terminate without causing the entire process to exit
- *pthread_kill (thr_kil)* – sends a signal to a specified thread
- *pthread_join (thr_join)* – causes the calling thread to wait for the specified thread to exit
- *pthread_self (thr_self)* – returns the caller's identity

POSIX vs Solaris 2 (2) – Old Book

Description	POSIX	Solaris 2
Thread Management	pthread_create pthread_exit pthread_kill pthread_join pthread_self	thr_create thr_exit thr_kill thr_join thr_self
Mutual exclusion	pthread_mutex_init pthread_mutex_destroy pthread_mutex_lock pthread_mutex_trylock pthread_mutex_unlock	mutex_init mutex_destroy mutex_lock mutex_trylock mutex_unlock
Condition variables	pthread_cond_init pthread_cond_destroy pthread_cond_wait pthread_cond_timewait pthread_cond_signal pthread_cond_broadcast	cond_init cond_destroy cond_wait cond_timewait cond_signal cond_broadcast

Thread Management Functions – New Book

POSIX function	description
<code>pthread_cancel</code>	terminate another thread
<code>pthread_create</code>	create a thread
<code>pthread_detach</code>	set thread to release resources
<code>pthread_equal</code>	test two thread ID's for equality
<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

POSIX: *THR* Threads

POSIX: *THR* uses attribute objects to represent thread properties

- Properties such as stack size or scheduling policy are set for a thread attribute object
- Several threads can be associated with the same attribute object
- If a property of an object changes, the change is reflected in all threads associated with the object
- POSIX: *THR* threads offer a more robust method of thread cancellation and termination (than Solaris)

Sun Solaris 2 Threads

- Solaris threads explicitly set properties of threads and other primitives
- Therefore, some calls have long lists of parameters for setting properties
- Solaris offers more control over how threads are mapped to processor resources (than POSIX)

Thread Management

- A thread has an ID, stack, execution priority, and starting address for execution (and perhaps scheduling and usage information)
- POSIX threads are referenced by an ID of type `pthread_t`
- A thread determines its ID by calling `pthread_self`
- Threads for a process share the entire address space for that process
- Threads are dynamic if they can be created at any time during execution
- POSIX: *THR* creates threads dynamically with `pthread_create` (creates thread and places it in the ready queue)

pthread_create

SYNOPSIS

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

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

POSIX: *THR*

- tid points to thread ID
- attr points to attributes of thread (NULL implies default attributes)
- start routine points to function thread calls when it begins execution
- start routine returns a pointer to void which is treated as exit status by pthread_join

pthread_exit/pthread_join

SYNOPSIS

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

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

```
int pthread_join(pthread_t thread, void **value_ptr);
```

POSIX.1*THR*

- pthread_exit terminates the calling thread
- The value_ptr parameter is available to a successful pthread_join
- However, the pthread_exit value_ptr parameter points to data that exists after the thread exits, so it cannot be allocated as an automatic local variable

PTHREAD Example

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <pthread.h>
void main(void)
{ pthread_t copy_tid;
  int myarg[2];
  int error;
  void *copy_file(void *arg);
  if ((myarg[0] = open("my.in", O_RDONLY)) == -1)
    perror("Could not open my.in");
  else if ((myarg[1] = open("my.out",
    O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR)) == -1)
    perror("Could not open my.out");
  else if (error=pthread_create(&copy_tid, NULL, copy_file,
    (void *)myarg))
    fprintf(stderr, "Thread creation was not successful: %s\n",
      strerror(error));
}
```

PTHREAD Example Analysis

- `copy_tid` holds the ID of the created thread
- `copy_file` is the name of the function the thread executes
- `myarg` is a pointer to the parameter value to be passed to the thread function (contains file descriptors for `my.in` and `my.out`)

copy_file – Top

```
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <errno.h>

#define BUFFERSIZE 100

void *copy_file(void *arg)
{
    int infile;
    int outfile;
    int bytes_read = 0;
    int bytes_written = 0;
    int *bytes_copied_p;
    char buffer[BUFFERSIZE];
    char *bufp;
```

copy_file – Middle

```
/* open file descriptors for source and destination files */
infile = *((int*)(arg));
outfile = *((int*)(arg) + 1);
if ((bytes_copied_p = (int*)malloc(sizeof(int))) == NULL)
    pthread_exit(NULL);
*bytes_copied_p = 0;

for ( ; ; ) {
    bytes_read = read(infile, buffer, BUFFERSIZE);
    if ((bytes_read == 0) || ((bytes_read < 0) && (errno != EINTR)))
        break;
    else if ((bytes_read < 0) && (errno == EINTR))
        continue;
    bufp = buffer;
    while (bytes_read > 0) {
        bytes_written = write(outfile, bufp, bytes_read);
        if ((bytes_written < 0) && (errno != EINTR))
            break;
    }
}
```

copy_file – Bottom

```
else if (bytes_written < 0)
    continue;
    *bytes_copied_p += bytes_written;
    bytes_read -= bytes_written;
    bufp += bytes_written;
}
if (bytes_written == -1)
    break;
}
close(infile);
close(outfile);
pthread_exit(bytes_copied_p);
}
```

What if malloc fails? – On pthread_join, bytes_copied_p is NULL and program crashes when it tries to dereference pointer (check for NULL pointer)

What if errno is global? macro?

Multi-FD Copier – Upper Top

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <pthread.h>
#define MAXNUMCOPIERS 10
#define MAXNAMESIZE 80

void *copy_file(void *arg);
```

Multi-FD Copier – Lower Top

```
void main(int argc, char *argv[])
{ pthread_t copiertid[MAXNUMCOPIERS];
  int fd[MAXNUMCOPIERS][2];
  char filename[MAXNAME_SIZE];
  int numcopiers;
  int total_bytes_copied=0;
  int *bytes_copied_p;
  int error;
  int i;
  if (argc != 4) {
    fprintf(stderr, "Usage: %s infile outfile copiers\n", argv[0]);
    exit(1);
  }
  numcopiers = atoi(argv[3]);
  if (numcopiers < 1 || numcopiers > MAXNUMCOPIERS) {
    fprintf(stderr, "%d invalid number of copiers\n", numcopiers);
    exit(1);
  }
```

Multi-FD Copier – Upper Bottom

```
for (i = 0; i < numcopiers; i++) {
    sprintf(filename, "%s.%d", argv[1], i);
    if ((fd[i][0] = open(filename, O_RDONLY)) < 0) {
        fprintf(stderr, "Unable to open copy source file %s: %s\n",
                filename, strerror(errno));
        continue;
    }
    sprintf(filename, "%s.%d", argv[2], i);
    if ((fd[i][1]=
        open(filename, O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR)) < 0) {
        fprintf(stderr,
            "Unable to create copy destination file %s: %s\n",
            filename, strerror(errno));
        continue;
    }
    if (error=pthread_create(&copiertid[i], NULL, copy_file,
        (void *)fd[i]))
        fprintf(stderr, "Could not create thread %d: %s\n",
            i, strerror(error)); }    /* wait for copy to complete */
```

Multi-FD Copier – Lower Bottom

```
for (i = 0; i < numcopiers; i++) {
    if (error=pthread_join(copiertid[i], (void *)&(bytes_copied_p)))
        fprintf(stderr, "No thread %d to join\n",i);
    else {
        printf("Thread %d copied %d bytes from %s.%d to %s.%d\n",
            i, *bytes_copied_p, argv[1], i, argv[2], i);
        total_bytes_copied += *bytes_copied_p;
    }
}
printf("Total bytes copied = %d\n", total_bytes_copied);
exit(0);
}
```

Bad Copier – Upper Top

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <pthread.h>

void *copy_file(void *arg);
#define MAXNUMCOPIERS 10
#define MAXNAMESIZE 80
```

Bad Copier – Lower Top

```
void main(int argc, char *argv[])
{ pthread_t copiertid[MAXNUMCOPIERS];
  int fd[2];
  char filename[MAXNAMESIZE];
  int numcopiers;
  int total_bytes_copied=0;
  int *bytes_copied_p;
  int error;
  int i;
  if (argc != 4) {
    fprintf(stderr, "Usage: %s infile_name outfile_name copiers\n",
      argv[0]);
    exit(1);
  }
  numcopiers = atoi(argv[3]);
  if (numcopiers < 1 || numcopiers > MAXNUMCOPIERS) {
    fprintf(stderr, "%d invalid number of copiers\n", numcopiers);
    exit(1);
  }
}
```

Bad Copier – Upper Bottom

```
for (i = 0; i < numcopiers; i++) {
    sprintf(filename, "%s.%d", argv[1], i);
    if ((fd[0] = open(filename, O_RDONLY)) < 0) {
        fprintf(stderr, "Unable to open copy source file %s: %s\n",
                filename, strerror(errno));
        continue;
    }
    sprintf(filename, "%s.%d", argv[2], i);
    if ((fd[1] =
        open(filename, O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR)) < 0) {
        fprintf(stderr,
            "Unable to create copy destination file %s: %s\n",
            filename, strerror(errno));
        continue;
    }
    if (error=pthread_create(&copiertid[i], NULL, copy_file,
        (void *)fd))
        fprintf(stderr, "Could not create thread %d: %s\n",
            i, strerror(error)); } /* wait for copy to complete */
```

Bad Copier – Lower Bottom

```
for (i = 0; i < numcopiers; i++) {
    if (error=pthread_join(copiertid[i], (void **)&(bytes_copied_p)))
        fprintf(stderr, "No thread %d to join: %s\n",
                i, strerror(error));
    else {
        printf("Thread %d copied %d bytes from %s.%d to %s.%d\n",
            i, *bytes_copied_p, argv[1], i, argv[2], i);
        total_bytes_copied += *bytes_copied_p;
    }
}
printf("Total bytes copied = %d\n", total_bytes_copied);
exit(0);
}
```

A different pair of file descriptors is opened for each thread, but the fd array is reused for each thread – If a sleep(5) is placed after the pthread_create, threads will probably be able to complete before a conflict occurs

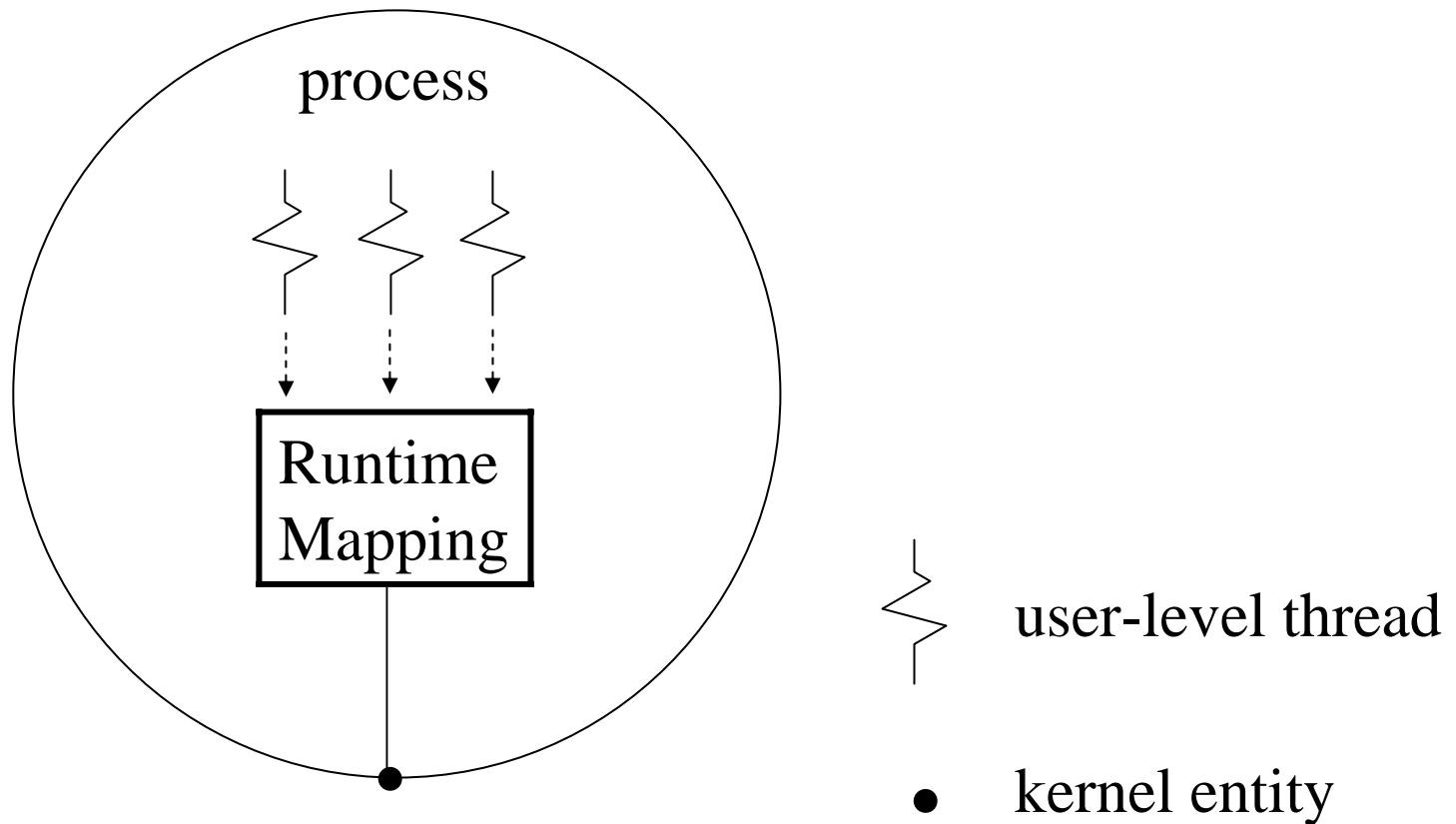
User-Level Thread Advantages

- Run on top of the existing operating system
- Compete among themselves for the resources allocated to a process
- Threads are scheduled by a run-time thread system that is part of the process code
- Each library/system call is enclosed by a “jacket” – jacket code calls the runtime system
- read and sleep could be a problem because they cause the process to block – the potentially blocking code is replaced in the jacket by non-blocking code
- If the code does not block, do the call right away – if the code does block, add it to a list to do later and pick another thread to run
- User-level threads have low overhead

User-Level Threads - Disadvantages

- Disadvantages
 - It relies on threads to allow the thread runtime system to regain control
 - CPU-bound thread rarely performs system/library calls preventing runtime system from regaining control to schedule other threads
 - Programmer must avoid lockout by explicitly forcing CPU-bound threads off CPU
 - Can share only resources allocated to encapsulating process – Therefore, they can only run on one processor (user-level threads are not good in a multi-processor system)

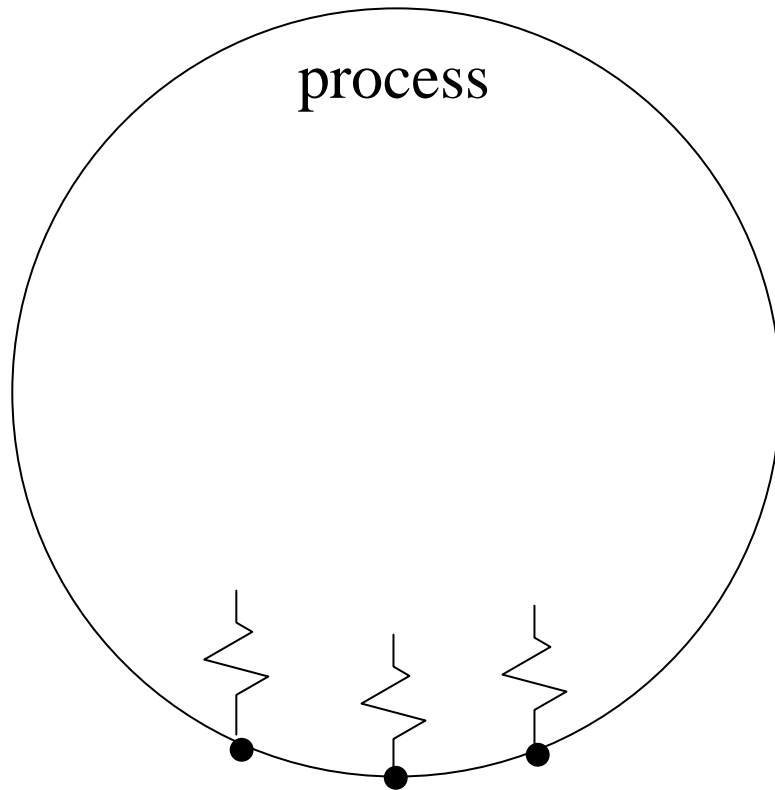
User-Level Threads



Kernel-Level Threads

- The kernel schedules each thread
- Threads compete for resources just like processes do
- Scheduling threads is more expensive – almost as expensive as scheduling processes
- Kernel-level threads can take advantage of multiple processors

Kernel-Level Thread Diagram

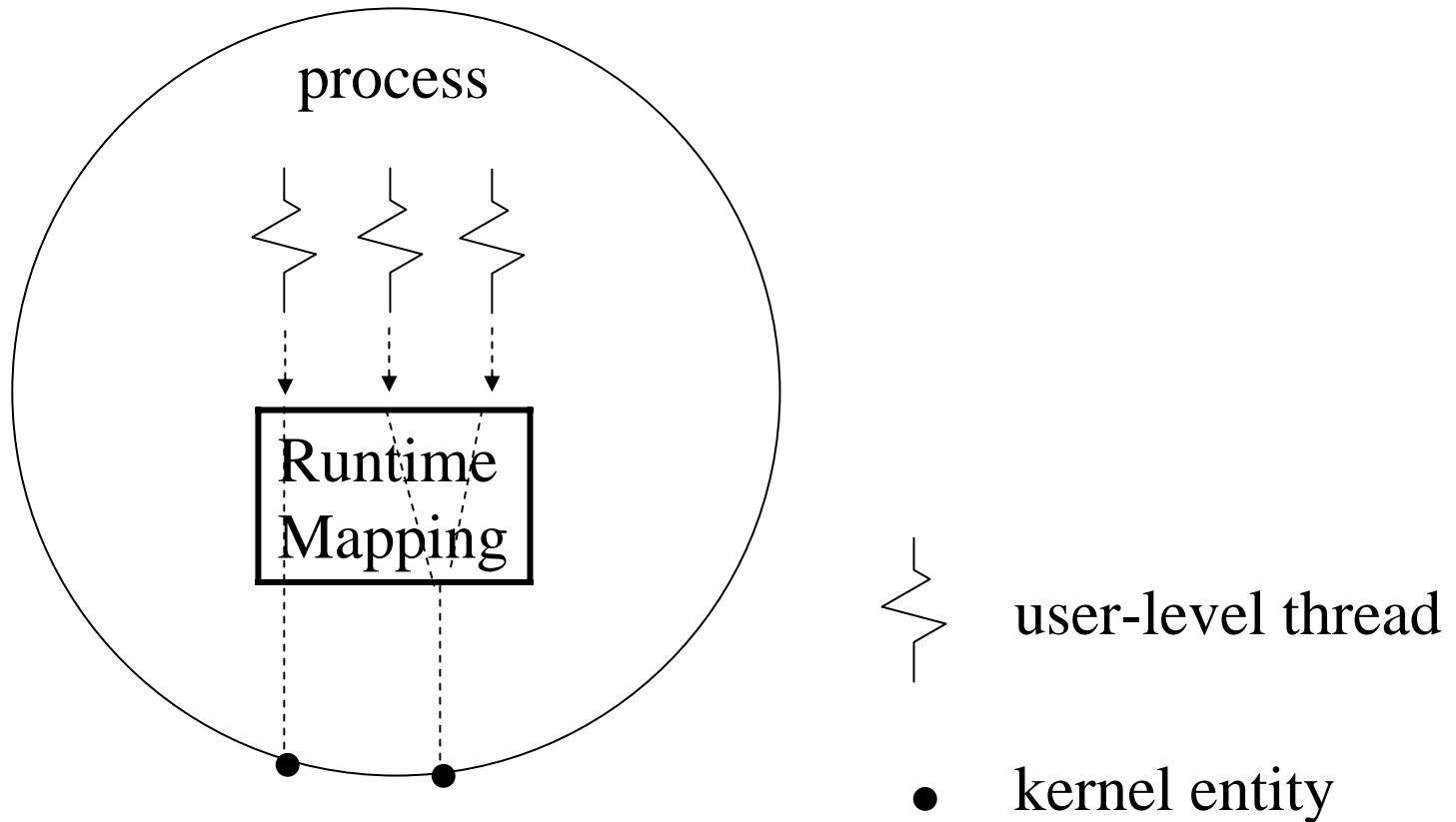


- kernel-level thread

Hybrid Threads

- Hybrid threads have the advantages of both user and kernel-level threads
- User writes program in terms of user-level threads and then specifies how many kernel-schedulable entities are associated with the process
- User-level threads are mapped into kernel-schedulable entities
- Sun Solaris calls the kernel-schedulable entities lightweight processes

Hybrid Thread Diagram



POSIX: *THR* Threads

- POSIX: *THR* supports hybrid threads
- Schedules threads and kernel entities
 - Threads are analogous to user-level threads
 - Kernel entities are scheduled by the kernel
 - Thread library decides how many kernel entities it needs and how to map them
- *Thread Scheduling Contention Scope*
 - Gives programmer control over how kernel entities are mapped to threads
 - *PTHREAD_SCOPE_PROCESS* contend for process resources
 - *PTHREAD_SCOPE_SYSTEM* contend for system resources

Solaris 2 Terminology

- *Unbound Thread* – User-level thread
- *Bound Thread* – Kernel-level thread (because it is bound to a lightweight process)
- *fork()* – Cost of creation of entire process
- *synchronization* – Two threads synchronizing with semaphores

Solaris 2.3 Service Times

Operation	Microseconds
Unbound thread create	52
Bound thread create	350
fork()	1700
Unbound thread synchronize	66
Bound thread synchronize	390
Between process synchronize	200

POSIX:*THR* Thread Attributes

- POSIX:*THR* takes an object-oriented approach to representation and assignment of thread properties
- Each POSIX:*THR* thread has an associated attribute object representing its properties
- An attribute object can be associated with multiple threads
- There are functions to create, configure, and destroy attribute objects
- When a property of an attribute object changes, all entities in the group have the new property
- Thread attribute objects are of type `pthread_attr_t`

POSIX.1c Attribute Objects

Property	Function
Initialization	<code>pthread_attr_init</code> <code>pthread_attr_destroy</code>
Stack Size	<code>pthread_attr_setstacksize</code> <code>pthread_attr_setstackaddr</code>
Detach State	<code>pthread_attr_setdetachstate</code> <code>pthread_attr_getdetachstate</code>
Scope	<code>pthread_attr_setscope</code> <code>pthread_attr_getscope</code>
Inheritance	<code>pthread_setinheritsched</code> <code>pthread_getinheritsched</code>
Schedule Policy	<code>pthread_attr_setschedpolicy</code> <code>pthread_attr_getschedpolicy</code>
Schedule Parameters	<code>pthread_attr_setschedparam</code> <code>pthread_attr_getschedparam</code>

Attribute Object – Example

```
...
#include <schedule.h>
#include <pthread.h>
#define HIGHPRIORITY 10
pthread_attr_t my_tattr;
pthread_t my_tid;
struct sched_param param;
int fd;
if (pthread_attr_init(&my_tattr))
    perror("Could not initialize thrad attribute object");
else if (pthread_create(&my_tid,&my_tattr, do_it, (void *)&fd))
    perror("Could not create copier thread");
else if (pthread_attr_getschedparam(&my_tattr, &param))
    perror("Could not get scheduling parameters");
else { param.sched_priority = HIGHPRIORITY;
    if (pthread_attr_setschedparam(&my_tattr, & param))
        perror ("could not set priority"); }
```