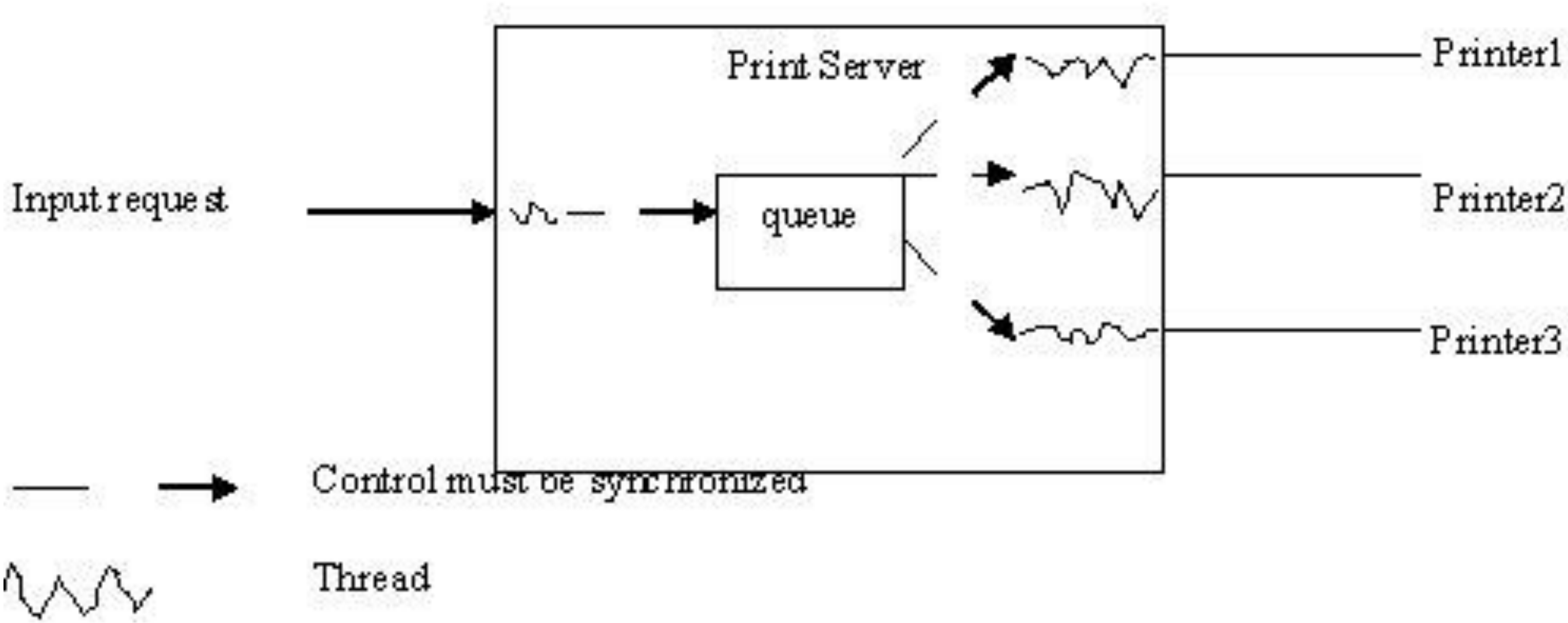
CSC552 – Advanced UNIX Programming

Threads

Dr. L. Frye
Kutztown University

Printer Server



UNIX Threads

- Exists within process and uses process resources
- Independent flow of control
- Share resources with other threads
- Dies with parent process

Process Structure

- Memory map (virtual addresses)
- File descriptor table
- Signal descriptor table
- User IDs
- Current working directory

Thread Maintains Own

- Registers
  - Stack Pointer
  - Program Counter
- Scheduling properties
- Set of pending and block signals
- Thread specific data

Process Structure

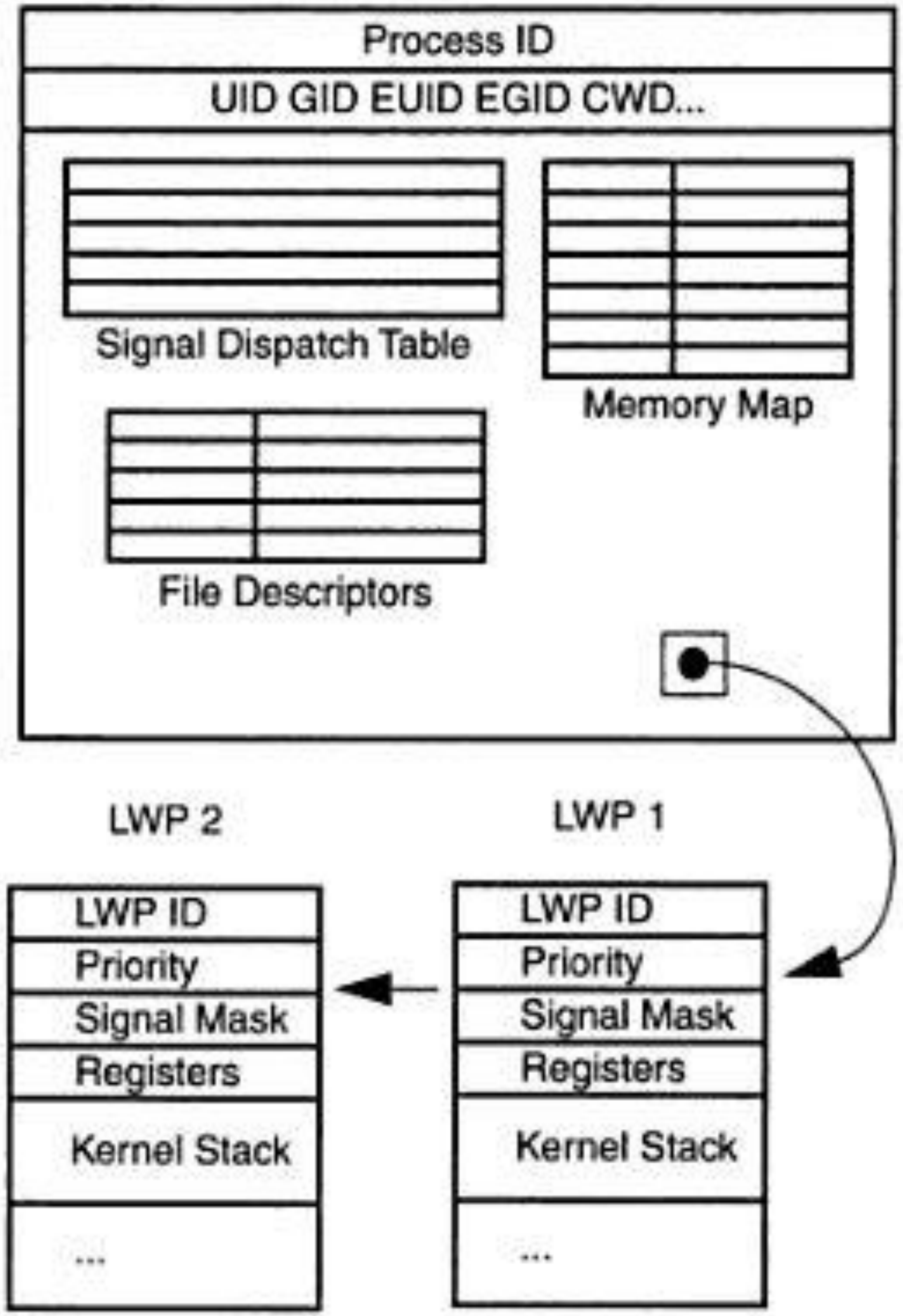
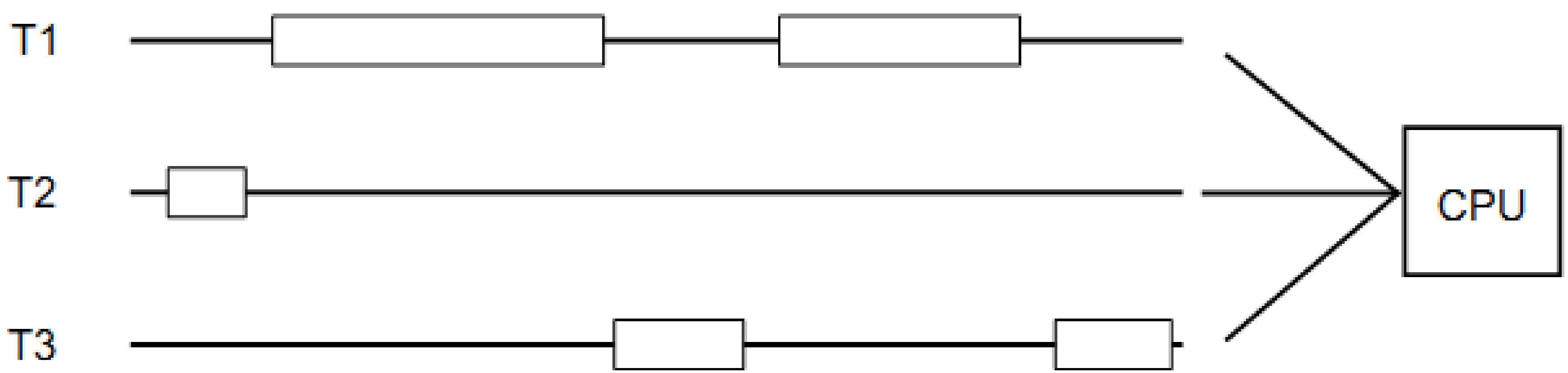


Image from "Multithreaded Programming with Pthreads by Lewis & Berg

Context-Switch

- Process
  - Registers
  - Virtual memory
  - Some other process state
- Threads
  - Registers

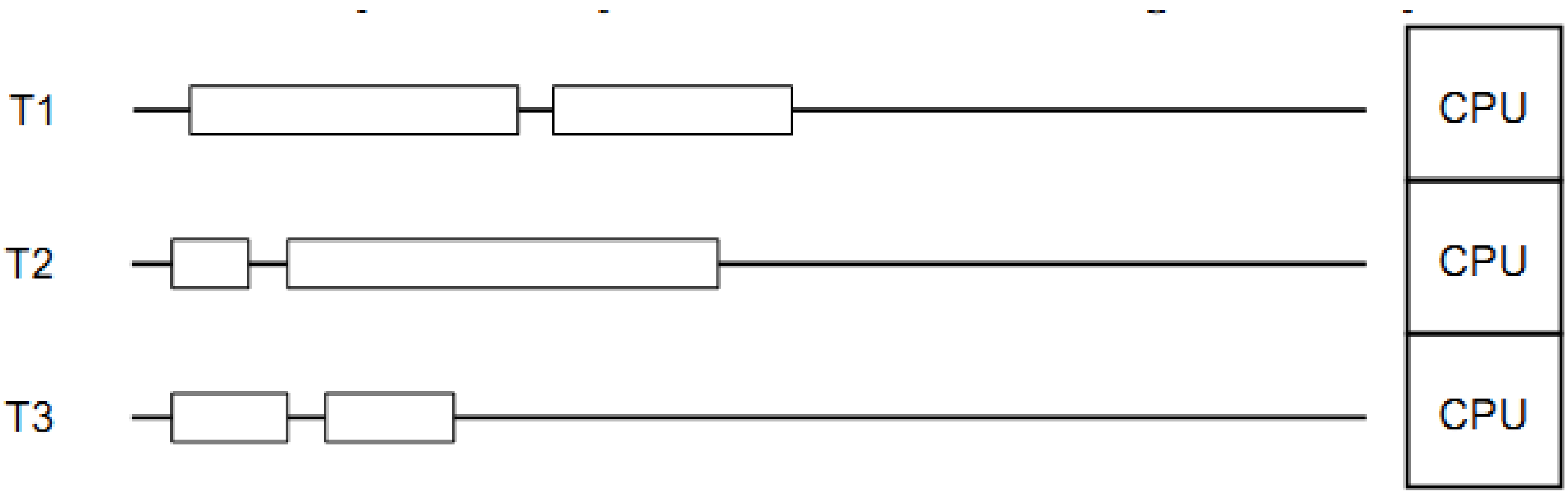
Concurrency



Three Threads Running Concurrently on One CPU

Image from "Multithreaded Programming with Pthreads by Lewis & Berg

Parallelism



Three Threads Running in Parallel on Three CPUs

Image from “Multithreaded Programming with Pthreads by Lewis & Berg

Performance

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
IBM332 MHz 604e 4 CPUs/node 512 MB Memory AIX 4.3	92.4	2.7	105.3	8.7	4.9	3.9
IBM 375 MHz POWER3 16 CPUs/node 16 GB Memory AIX 5.1	173.6	13.9	172.1	9.6	3.8	6.7
INTEL 2.2 GHz Xeon CPU/node 2 GB Memory RedHat Linux 7.3	17.4	3.9	13.5	5.9	0.8	5.3

Suitable Tasks

- Block for long waits
- Use many CPU cycles
- Must respond to asynchronous events
- Lesser or greater importance
- Performed in parallel

Common Models

- Manager/worker
- Pipeline
- Peer

POSIX Threads

- pthreads
- Return
  - 0
  - Error number
- Thread ID
  - pthread_t
- Basic Functions
  - pthread_self()
  - pthread_equal()

Thread Creation

- pthread_create()
- Main – single, default thread

- After a thread has been created, how do you know when it will be scheduled to run by the OS?

Create Example

-
```
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);
    ......
}
```

- What is wrong with this code fragment?
- How can it be corrected?

More Examples

- threads/hello.c
- threads/hello_struct.c

Freeing space

- Return value
  - Return function
  - Free space
- Parameter
  - Do not reuse
  - Separate variable

- threads/copymultiple.c
- Page 429, Ex 12.13

Thread Termination

- Thread returns from starting routine
- Calls pthread_exit()
- Cancelled by another thread calling pthread_cancel()

- pthread_detach()

Joinable Threads

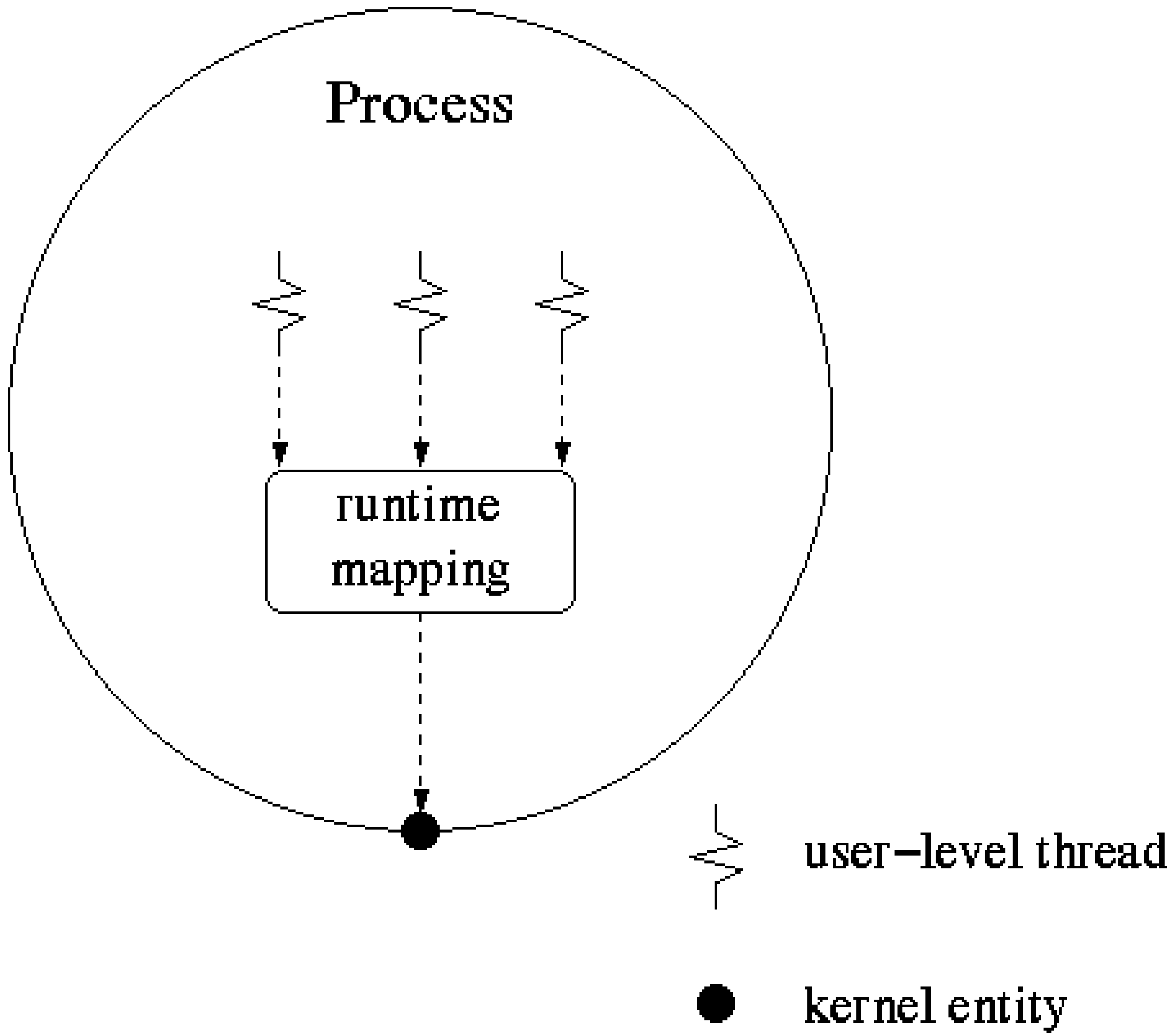
- Detachable vs. Nondetachable
- Suspend execution of calling thread until joined thread terminates
  - pthread_join()

- Single thread exit
  - pthread_exit()

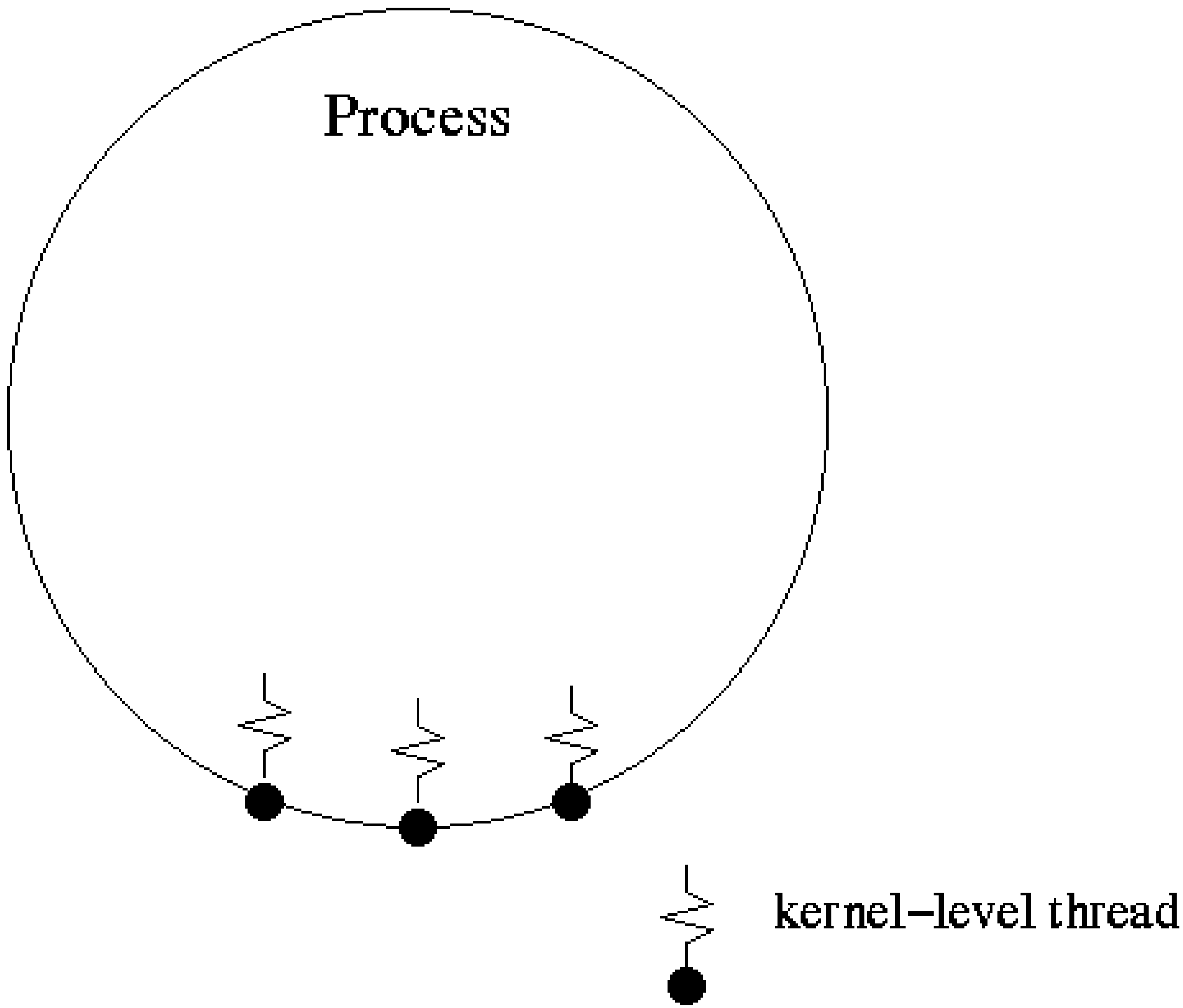
Cancel a Thread

- pthread_cancel()
- pthread_setcanceltype()
- pthread_testcancel()

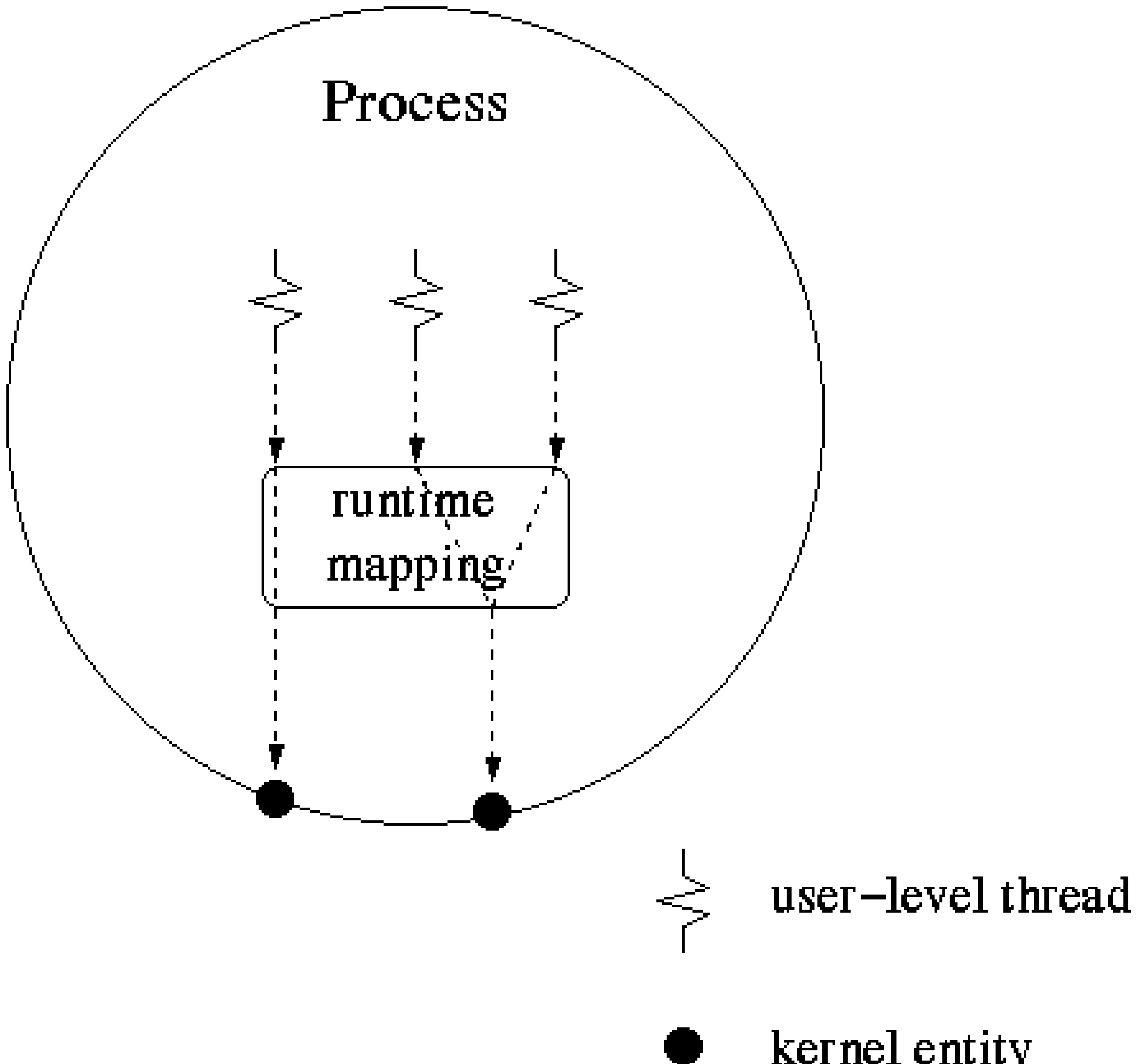
User-level Threads



Kernel-level Threads



Hybrid Threads



Mutex

- Mutual exclusion
- Two states
  - Locked
  - Unlocked
- Synchronization
  - Critical sections
  - Shared resources

Typical Sequence

- Create and initialize
- Several threads attempt to lock the mutex
- Only one succeeds
- The owner thread performs some set of actions
- The owner unlocks the mutex
- Another thread acquires the mutex and repeats the process
- Finally the mutex is destroyed

Mutex Functions

- Type: pthread_mutex_t

- pthread_mutex_init()
- pthread_mutex_destroy()
- pthread_mutex_lock()
- pthread_mutex_trylock()
- pthread_mutex_unlock()

Cooperation

- An uncooperative thread can enter a critical section without acquiring a mutex lock.

- What are some ways to prevent this from happening?

Example

- threads/counter.c

- What are possible side affects or what can go wrong if the *count* variable is not protected by a mutes lock?

- What really happens when a variable is incremented?

Protect Library Functions

```
int randsafe(double *ranp) {
static pthread_mutex_t lock =
                    PTHREAD_MUTEX_INITIALIZER;
 int error;

 if (error = pthread_mutex_lock(&lock))
      return error;
 *ranp = (rand() + 0.5)/(RAND_MAX + 1.0);
 return pthread_mutex_unlock(&lock);
}
```

Wait for Condition

- Busy wait
- Mutex
  - Lock mutex
  - Test condition
  - If true, unlock mutex and exit loop
  - If false, suspend thread and unlock mutex

Condition Variables

- Associated with specific condition
- Atomic waiting operation

- Type: pthread_cond_t
- Initialize
  - PTHREAD_CONDITION_INITIALIZER
  - pthread_cond_init()
- Destroy
  - pthread_cond_destroy()

Condition Wait

- pthread_cond_wait()

- pthread_mutex_lock(&m);
  while (x != y)
  	pthread_cond_wait(&v, &m);
  /* modify x or y if necessary */
  pthread_mutex_unlock(&m);

Condition Signal

- pthread_cond_signal()
- pthread_cond_broadcast()

- pthread_mutex_lock(&m)
  x++;
  pthread_cond_signal(&v);
  pthread_mutex_unlock(&m);

- pthread_cond_timedwait()

Guidelines

- Acquire the mutex before testing the predicate
- Retest the predicate after returning from a pthread_cond_wait – Why?
- Acquire the mutex before changing any of the variables appearing in the predicate.
- Hold the mutex only for a short period of
- Release the mutex
  - explicitly – pthread_mutex_unlock()
  - implicitly – pthread_cond_wait()

Examples

- threads/condvar1.c
- threads/tbarrier.c
- threads/syncConditionVar.c

Signal Handling and Threads

- All threads in process share process's signal handlers.
- Each thread has its own signal mask.

- What does this mean?

Signals and Threads

- Signal Types
  - Asynchronous
  - Synchronous
  - Directed

- pthread_sigmask()