Processes

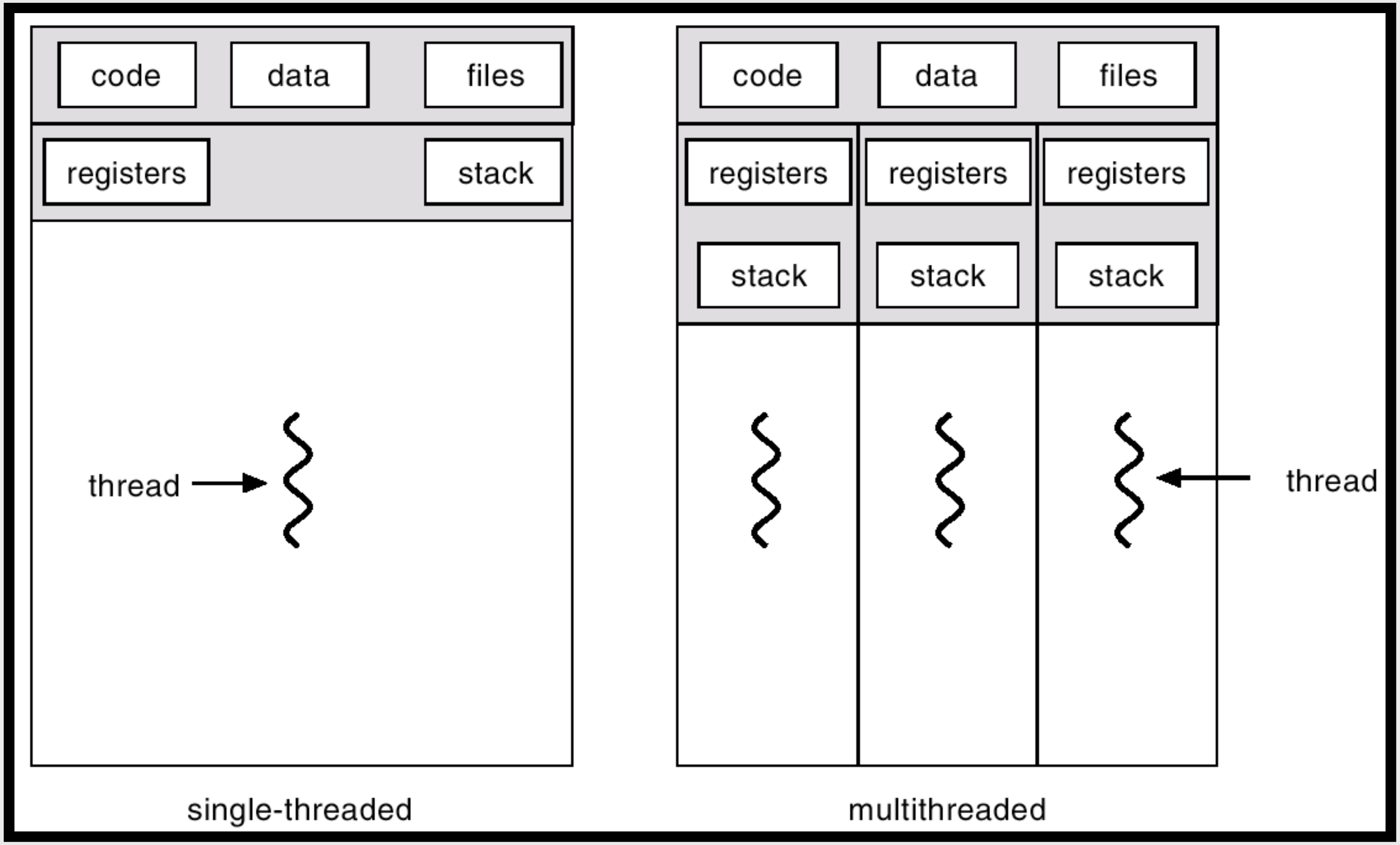
- “Heavy weight” control flow
  - Context switch
    - Non negligible cost
    - A lot of data to be saved/restored
    - Cause cache miss and TLB miss (see virtual memory lecture)

  - Isolated address space
    - Sharing data is painful
      - Shared memory
      - Message passing
      - …
  - Thread concept

Thread concept

- Lightweigh process
  - Light context
  - A shared part : memory, open file …
  - A private part : stack, registers, ...
- Schedulable execution context
  - More efficient context switch
- Ease the programming of concurent applications
  - Multiple execution flow in a same adress space
  - Immediate data sharing
- Simple programs use one thread per process
- But can also have multi-threaded programs
- Multiple threads running in same process's address space

Single-threaded et multi-threaded Process



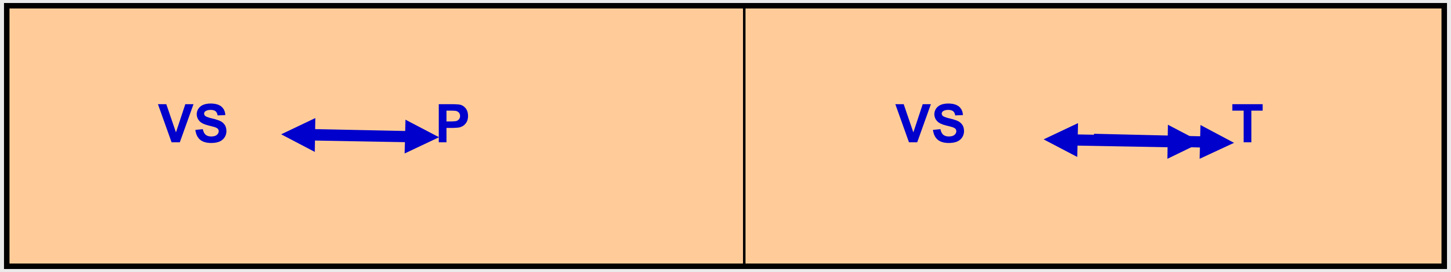
A.Sylberschatz

Why Thread

- Responsiveness
  - Do not block the whole program when only a part of it should be blocked
  - Allows program to overlap I/O and computation (same benefit as OS running emacs & gcc simultaneously)
- Resource sharing
  - Lighter-weight abstraction than processes (IPC, shmem)
  - All threads in one process share memory, file descriptors, etc
- Economy
  - Allocating memory, resources and context switching for process is costly
- Scalability
  - A single process can only use a single CPU at a time
  - Allows one process to use multiple CPUs or cores

Thread history

- **Adress Space**
- **Execution flow**
  - 2 notions tied together with the process concept
  - Threads dissociate these two notions



VS ⟷ P

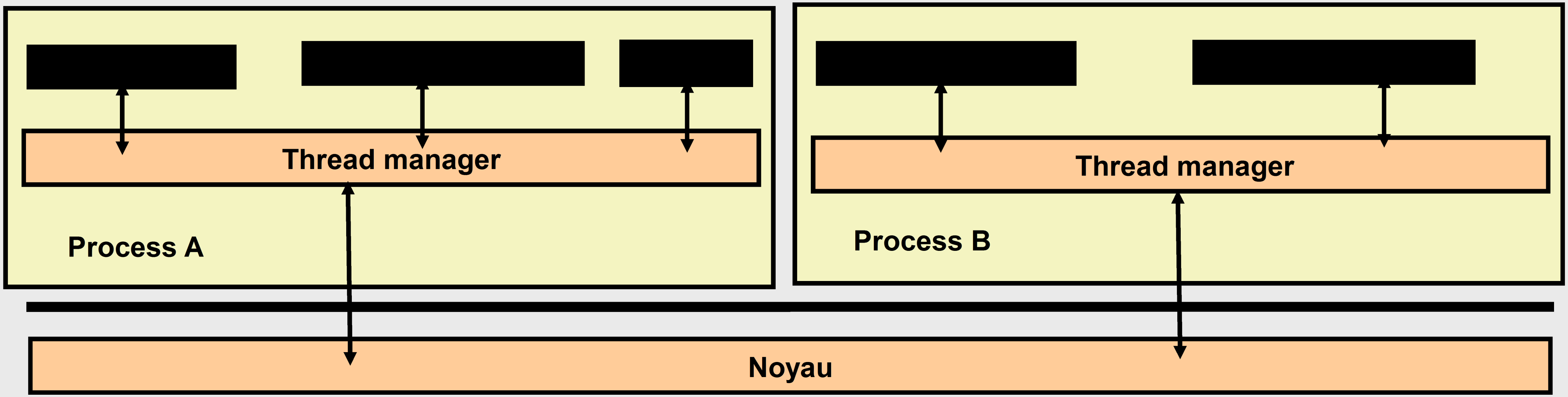
VS ⟷ T

- **Differents kind of thread**
  - User-level threads
  - Kernel-supported
  - Mixed solutions

User-level Threads

- **Implemented in a user level library**
- **Unmodified Kernel**
- **Threads and thread scheduler run in the same user process**

Examples: POSIX Pthreads, Mach C-threads, Solaris threads



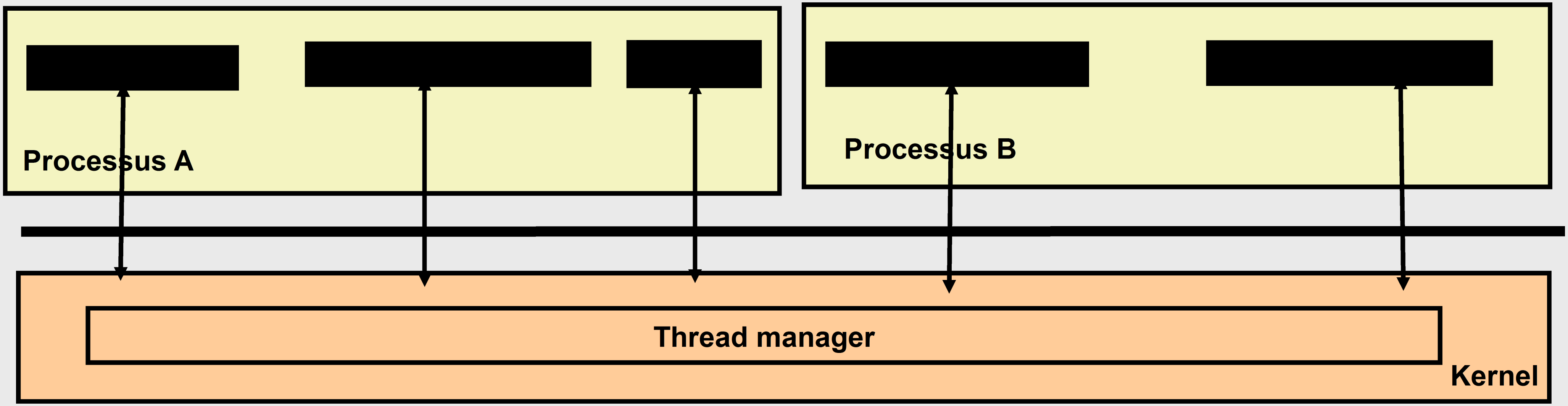
Advantages and disadvantage of User-level threads

- Parallelism (-)
  - No real parallelism between the threads within a process
- Efficiency (+)
  - Quick context switch
- Blocking system call (-)
  - The process is blocked in the kernel
  - All thread are blocked until the system call (I/O) is not terminated

Threads *kernel-supported*

- **Thread managed by the kernel**
  - Thread creation as a system call
- **When a thread is blocked, the processor is allocated to another thread by the kernel**

Examples: Windows NT/2000, Solaris, Tru64 UNIX, Linux



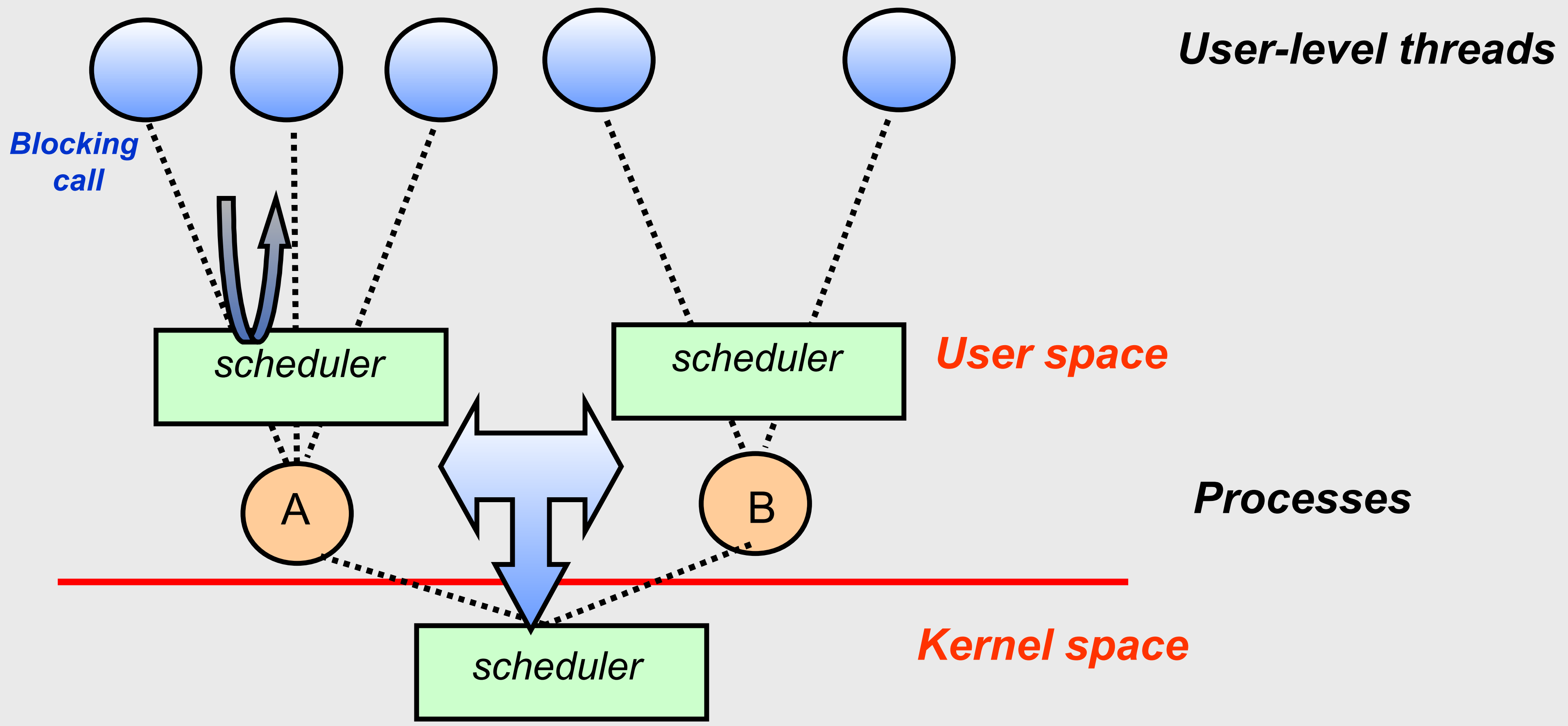
Kernel-supported pro and cons

- **Blocking system call (+)**
  - When a thread is blocked dur to a SVC call, the tread in the same virtual space are not
- **Real Parallelism (+)**
  - N threads in the same virtual space can run on K processors
- **Efficiency (-)**
  - More expensive context switch / user level threads
    - Every management operation goes throught the kernel
    - Require more memory

Hybrid solution for light context switch (sol 1)

- Take the best of the above implementations
  - Light context switch of user level thread
  - Avoid blocking all threads when invoking a system call
- Follows the principles of user level threads
  - Context switch managed at user level
- Kernel modification to manage system call to avoid blocking
  - When a thread uses a blocking svc, the kernel does not preempt the processor
  - Signals used to managed the end to the blocking svc
  - Two cooperating scheduler : user and system level

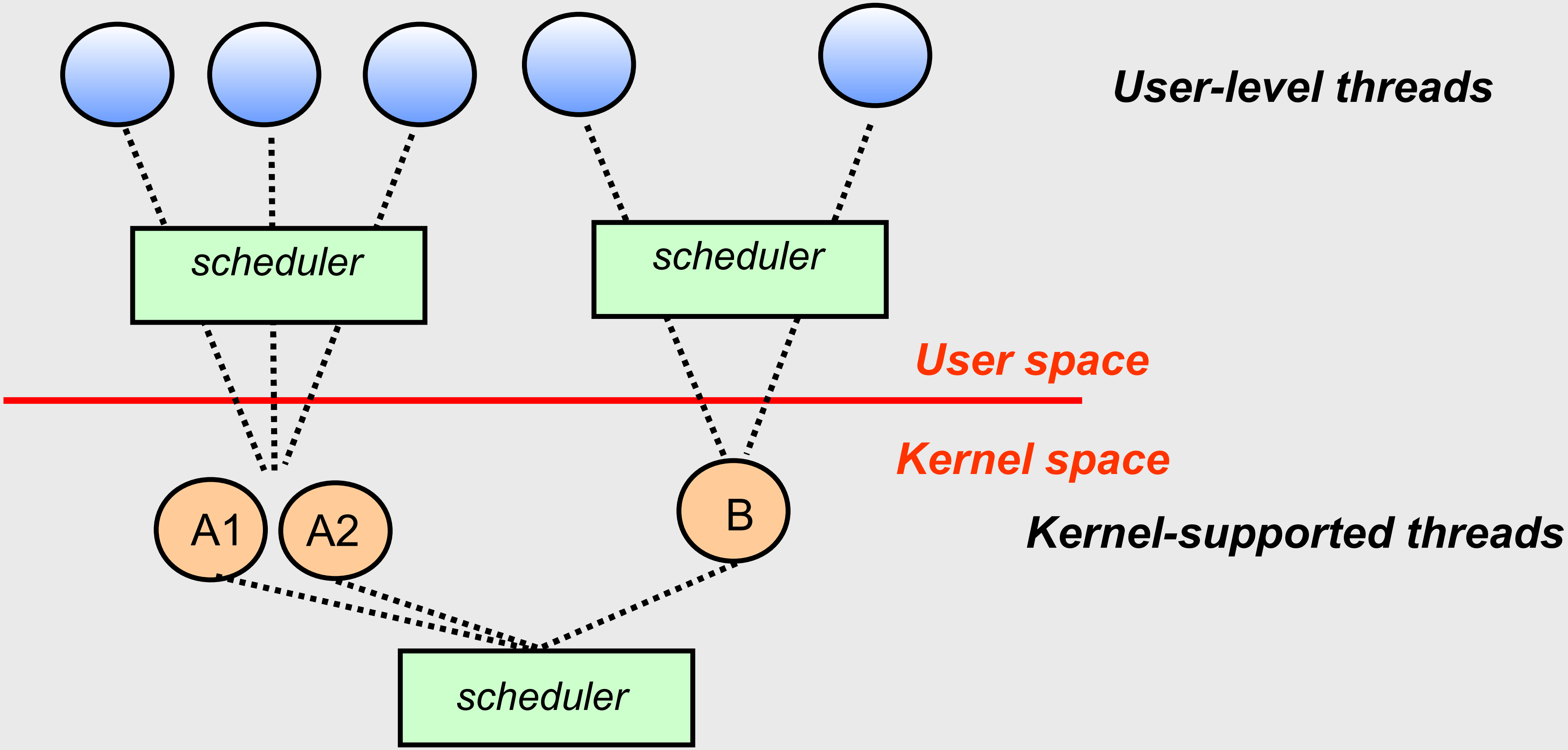
Hybrid solution for light context switch



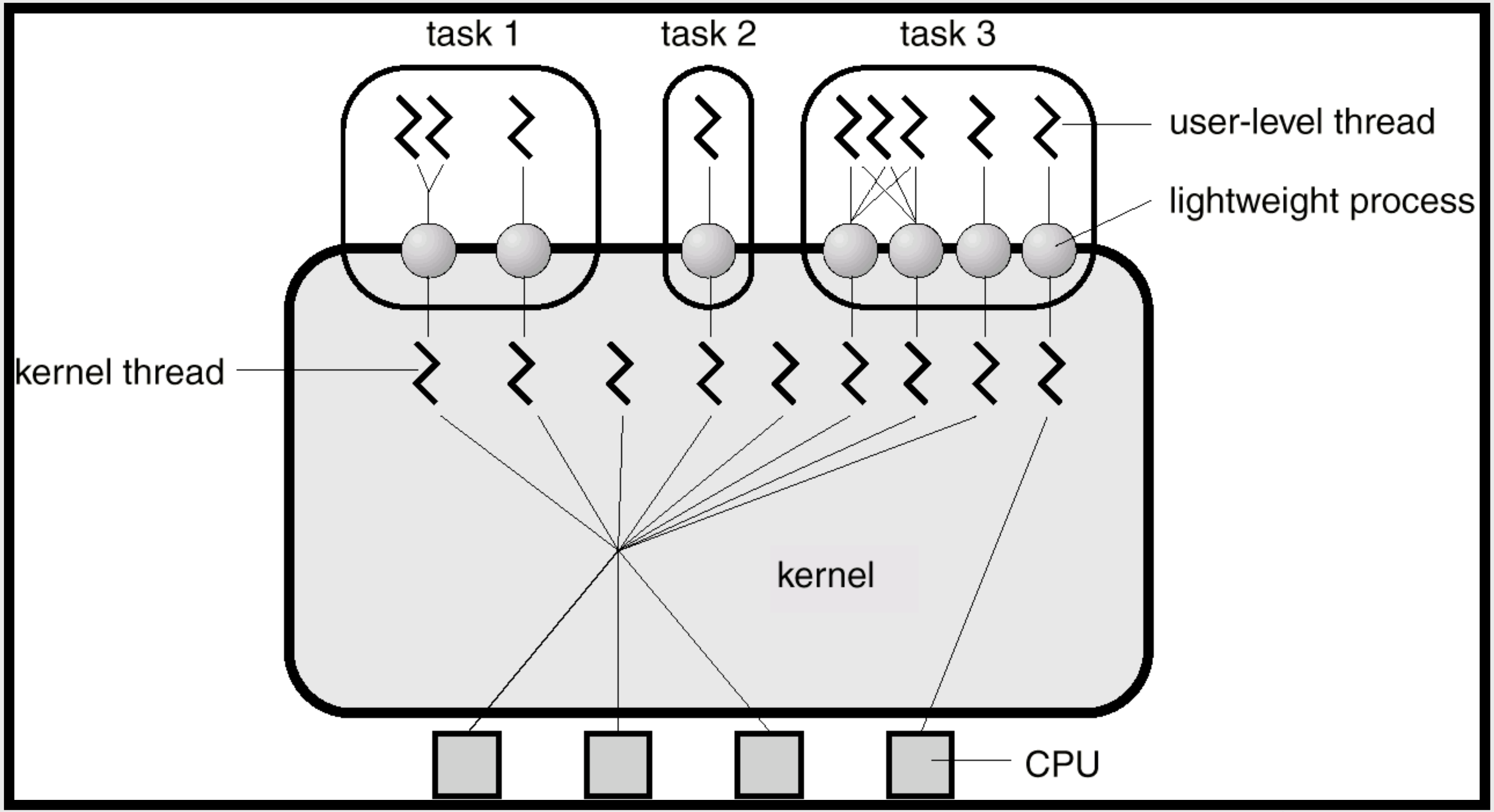
Hybrid solution for light context switch to enable real parallelism (sol 2)

- User threads implemented on kernel threads
  - Multiple kernel level threads per process
  - Thread creation, destruction still library fonctions
- Sometimes called n:m threading
  - Have n user threads per m kernel threads
  - (simple user level threads are n:1, kernel thread are 1:1)
- A pool of user threads mapped on a pool of kernel threads

Hybrid solution for light context switch to enable real parallelism (sol 2)



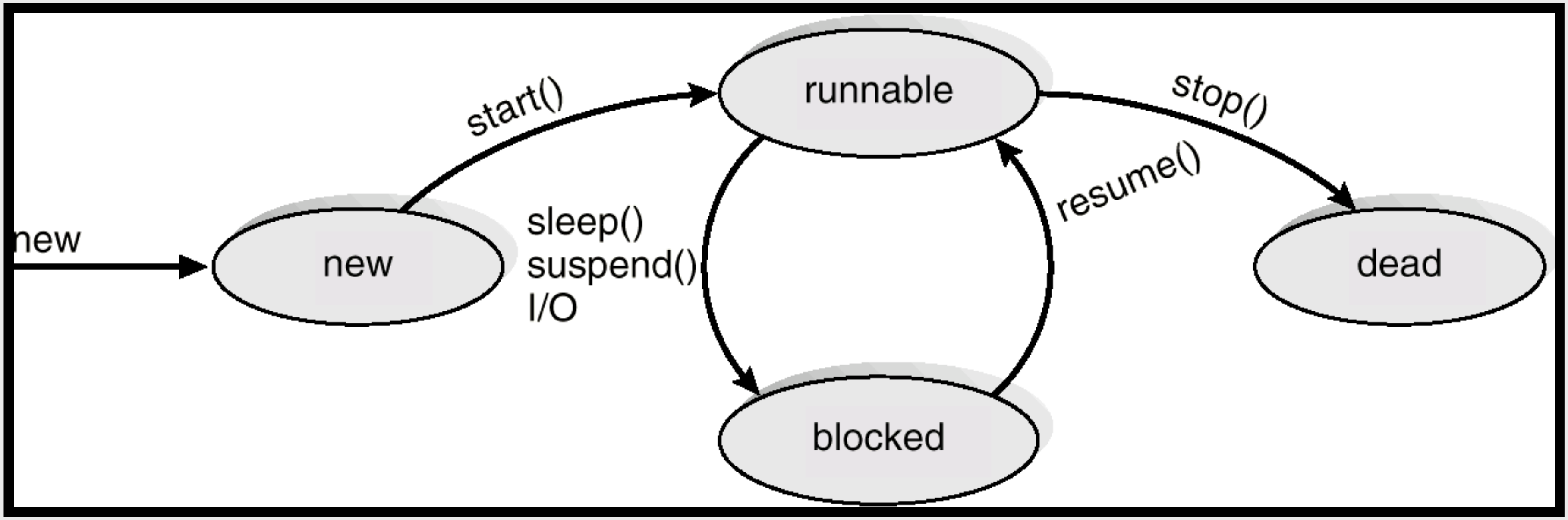
Tread model : many to many



A. Sylberschatz

Threads Java

- **User-level / Kernel-supported, depending on the JVM implementation**
- **Creation**
  - **Thread class extension**
  - **Implementation of the Runnable interface**



POSIX Threads : Pthreads API

- int thread create (tid, attr, void (*fn) (void *), void *);
  - Create a new thread, run fn with arg
- void thread exit ();
  - Destroy current thread
- void thread join (tid thread);
  - Wait for thread thread to exit
- Plus lots of support for synchronization [next week]

Posix thread creation

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

- Create a thread
- Run the start_fct with arg as arguments
- Tid : Id of the created thread

Example 1/2

```
#include <pthread.h>

void * ALL_IS_OK = (void *)123456789L;

char *mess[2] = { "thread1", "thread2" };

void * writer(void * arg)
{ int i, j;

  for(i=0;i<10;i++) {
    printf("Hi %s! (I'm %lx)\n", (char *) arg, pthread_self());
    j = 800000; while(j!=0) j--;
  }

  return ALL_IS_OK;
}
```

Example 2/2

```
int main(void)
{ void * status;
  pthread_t writer1_pid, writer2_pid;

   pthread_create(&writer1_pid, NULL, writer, (void *)mess[1]);
   pthread_create(&writer2_pid, NULL, writer, (void *)mess[0]);

   pthread_join(writer1_pid, &status);
   if(status == ALL_IS_OK)
      printf("Thread %lx completed ok.\n", writer1_pid);

   pthread_join(writer2_pid, &status);
   if(status == ALL_IS_OK)
      printf("Thread %lx completed ok.\n", writer2_pid);

   return 0;
}
```

Fork(), exec()

- What happens if one thread of a program calls fork()?
  - Does the new process duplicate all threads ? Or is the newprocess single-threaded ?
  - Some UNIX systems have chose to have two versions of fork()
- What happens if one thread of a program calls exec()?
  - Generally, the new program replace the entire process, including all threads.

Critical section problem

```
int count = 0;
void loop(void *ignored) {
  int i ;
  for (i=0 ; i<10 ; i++)  count++;
}
int main () {
  tid id = thread_create (&tid, NULL,loop, NULL);
  loop (); thread_join (id);
  printf("%d",count);
}
```

What is the output of this program ?

Critical section problem

- Need solution to critical section problem
    - Place count++ in critical section
- Protect critical sections from concurrent execution
    - n processes all competing to use some shared data
    - Each process has a code segment, called critical section, in which the shared data is accessed.
- Problem ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Desired properties

- Mutual Exclusion
  - Only one thread can be in critical section at a time
- Progress
  - Say no process currently in critical section (C.S.)
  - One of the processes trying to enter will eventually get in
- Bounded waiting
  - Once a thread T starts trying to enter the critical section, there is a bound on the number of times other threads get in

Mutual exclusion (next lecture)

- pthread_mutex_t : lock type
- pthread_mutex_init : lock init
- pthread_mutex_lock : lock the mutex
- pthread_mutex_unlock : unlock the mutex

```
pthread_mutex_t mon_mutex;
pthread_mutex_init(&mon_mutex,NULL);
pthread_mutex_lock(&mon_mutex);
… critical section
pthread_mutex_unlock(&mon_mutex);
//fin du programme
pthread_mutex_destroy(&mon_mutex);
```

Conditions (next lecture)

- `pthread_cond_t` : condition type
- `pthread_cond_init` : condition initialization
- `pthread_cond_wait` : block the thread on a condition and unlock the mutex
- `pthread_cond_signal` : wake the thread on the condition et relock the mutex

Beware the signaled thread do not take control immediately ...