Concurrency

Science

Computer Science

CS 450: Operating Systems

Michael Lee <lee@iit.edu>

ILLINOIS TECH | College of Computing

Agenda

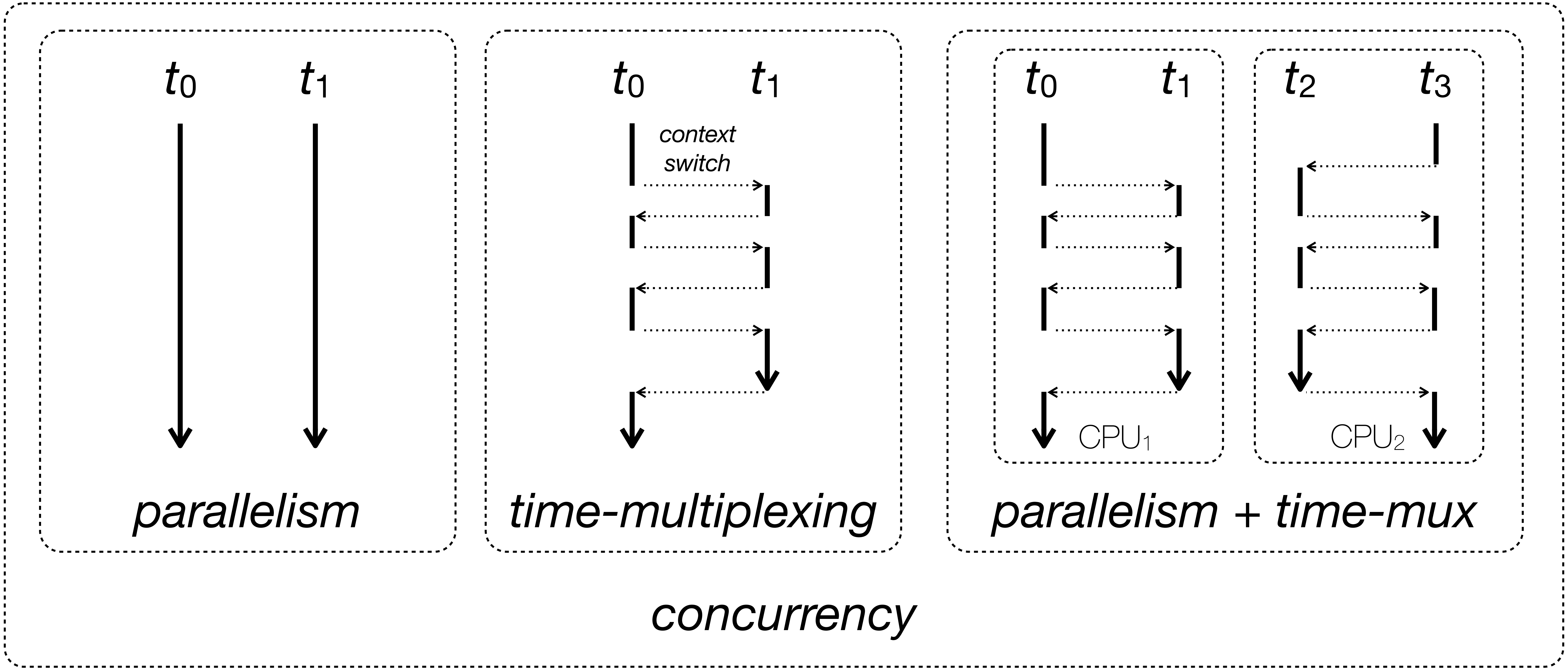
- Concurrency: what, why, how
  - Threads and Multithreading
- Parallelization and its limits
- Writing concurrent programs
  - Locks and locking strategies
  - Semaphores and synchronization

§ Concurrency: what, why, how

What is concurrency?

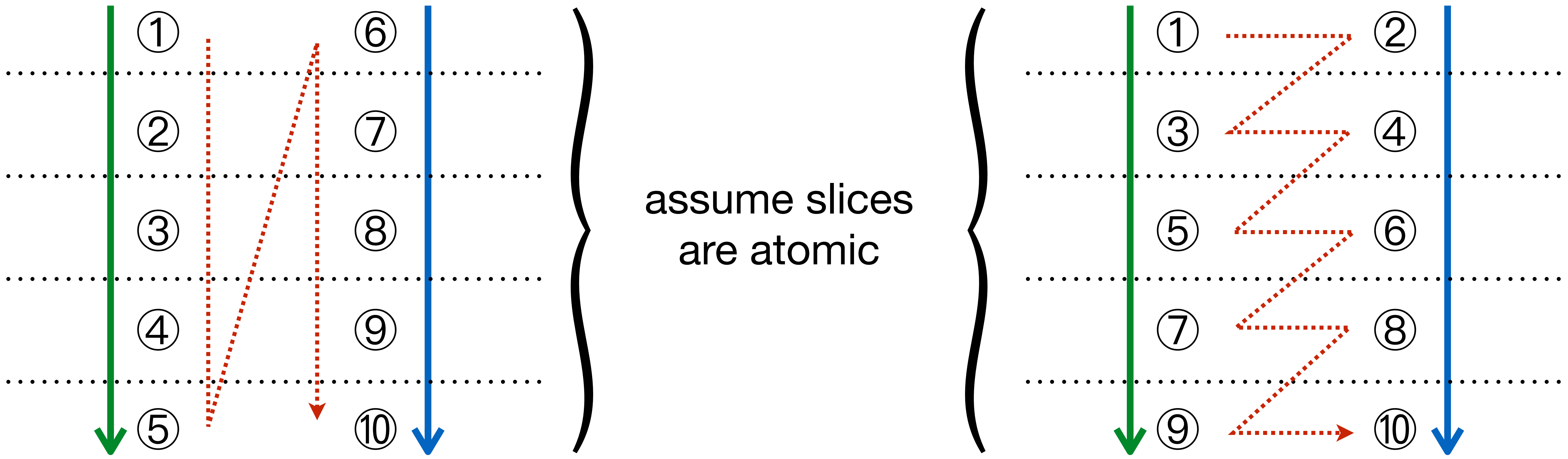
- Concurrency exists when two or more tasks overlap in their execution
- **Parallelism**, requiring multiple CPUs, is one way of realizing concurrency
  - e.g., tasks run at the same time on different CPUs
- Concurrency can also be achieved via **time-multiplexing**
  - e.g., via context switches on a single CPU
- Parallelism and time-multiplexing may coexist
  - e.g., N tasks running on M CPUs, $N > M > 1$

Concurrency and Parallelism



Non-determinism

- Both parallel and non-parallel forms of concurrency are *non-deterministic*
  - I.e., the execution order of different portions of the overlapping tasks is not pre-determined
- E.g., both orderings below are possible:



Process-level concurrency

- Multitasking OSes inherently support process-level concurrency
  - By default, processes run independently and may overlap in execution
- As we’ve seen, kernel runs each process in its own virtual sandbox
  - “Share-nothing” architecture: separate memory and control flow
  - Context switches triggered by traps & interrupts
  - Processes cannot easily interfere with each other!

e.g., Unix `fork`

- `fork` creates a child process, running concurrently with the parent
  - Same program (initially), but separate control flow and address space

```
int glob = 0;

main() {
    pid_t pid;
    for (int i=0; i<5; i++)
        if ((pid = fork()) == 0) {
            glob += 1;
            printf("Child %d glob = %d\n", i, glob);
            exit(0);
        } else
            printf("Parent created child %d\n", pid);
}
```

```
Parent created child 97447
Parent created child 97448
Parent created child 97449
Child 1 glob = 1
Parent created child 97450
Child 2 glob = 1
Parent created child 97451
Child 4 glob = 1
Child 3 glob = 1
Child 0 glob = 1
```

Single thread of execution

- Processes typically begin life with a *single thread of execution*
  - One path through the program (i.e., singular flow of control)
  - One stack (that reflects the active and preceding stack frames)
  - Blocking this thread (e.g., with I/O) blocks the entire process
- This model precludes *intra-process* concurrency
  - Why might we want more than one thread?

Intra-process concurrency

- There are many scenarios where support for concurrency within a process may come in handy. Generally, we might want to:

1. Improve CPU utilization
2. Improve I/O utilization
3. Improve performance via parallelization (most elusive!)

1. Improve CPU utilization

- E.g., consider interleaved but independent CPU & I/O operations:

```
while (1) {
    result = long_computation(); // CPU-bound operation
    update_log_file(result);     // blocks on I/O
}
```

  - Single threaded execution forces CPU-bound operation to wait for I/O to complete
  - Logically, should be able to start a new computation while logging the result from the previous loop

2. Improve I/O utilization

- E.g., consider multiple operations that block on unrelated I/O:

```
read_from_disk1(buf1);    // block for input
read_from_disk2(buf2);    // block for input
read_from_network(buf3); // block for input
process_input(buf1, buf2, buf3);  // process inputs
```

  - Single threaded execution forces I/O calls to take place sequentially — i.e., cannot start a request before the previous one completes
  - Would prefer to initiate I/O operations simultaneously!

3. Improve performance

- E.g., consider independent computations over large data set:

```
int A[DIM][DIM], // src matrix
    B[DIM][DIM], // src matrix
    C[DIM][DIM]; // dest matrix

/* C = A x B */
int i, j, k;
for (i=0; i<DIM; i++) {
    for (j=0; j<DIM; j++) {
        C[i][j] = 0;
        for (k=0; k<DIM; k++)
            C[i][j] += A[i][k] * B[k][j];
    }
}
```

each result cell can be computed independently!

Multiple threads

- In each preceding scenario, we could use multiple threads within a single process, each of which *runs concurrently* and *blocks independently*
- Each thread of execution should:
  - Share the address space of other threads in the same process
  - Maintain its own thread-specific state and data

Global (shared)

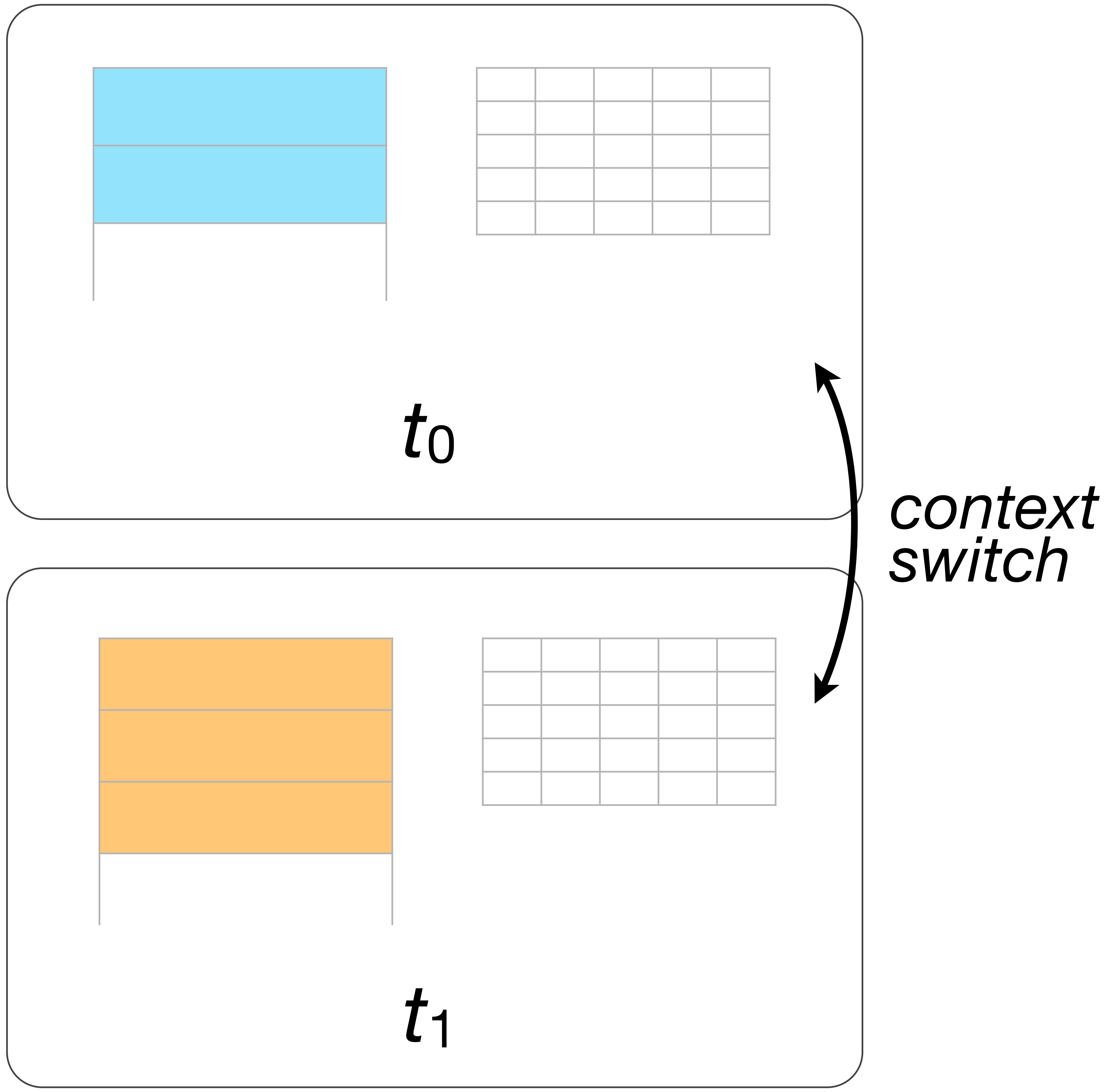
Code

Data

Thread-local

Stack

Regs



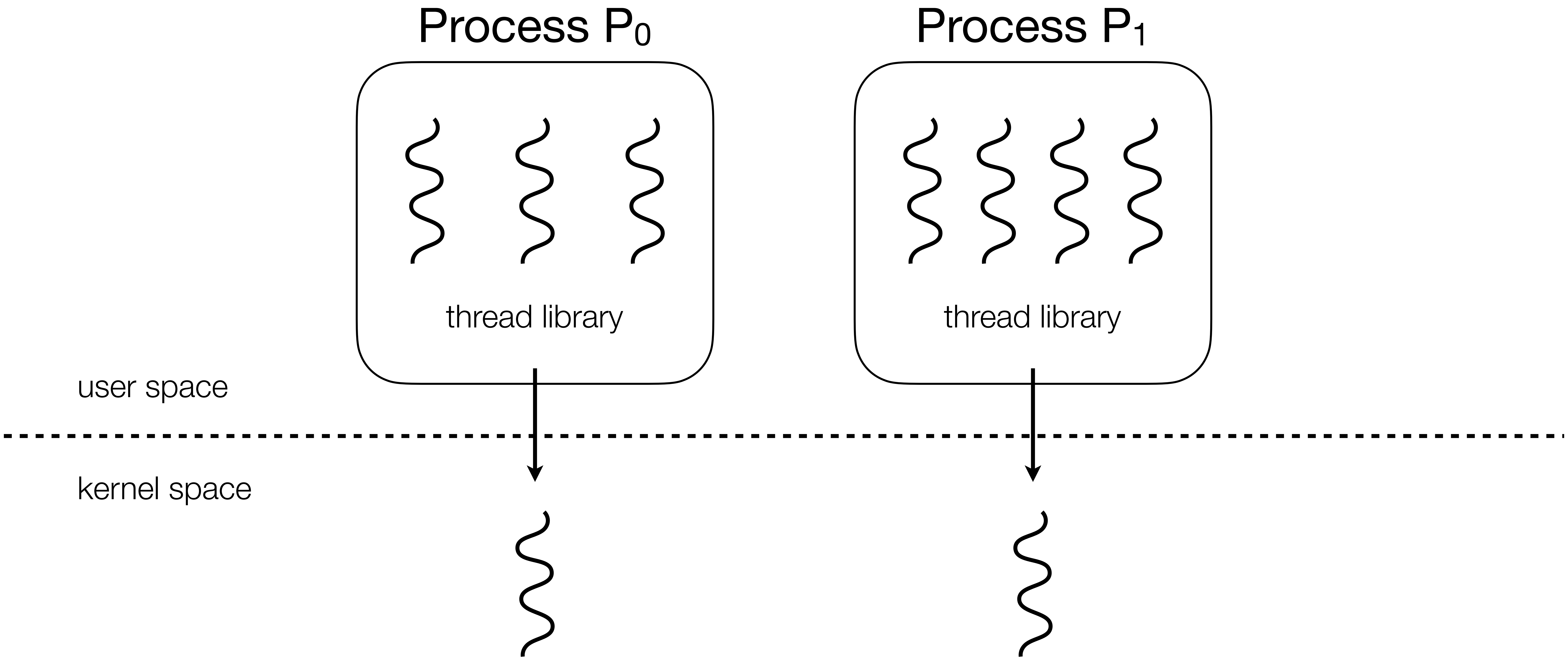
Implementing threads

- Each thread requires:
  - a stack
    - for maintaining function activation records, local variables, etc.
  - a thread control block (thread-specific analog of the PCB)
    - PC, SP, and other register values; TID; state and accounting info, etc.
  - CPU time (as allocated by the scheduler)
- Threads can be implemented at either the user or kernel level

User-level (aka green) threads

- Invisible to the kernel, which continues to schedule each process as a single-thread of execution
- Thread data/metadata is tracked by the process (user-level code)
  - Allocates stacks and TCBs as user-space data structures
- Thread scheduling and context switches are triggered by system timers (e.g., `SIGALARM` on Unix)
  - Alternatively, can implement purely cooperative thread (aka “fiber”) multitasking — only context switch on manual “yield” call

N:1 mapping of user→kernel threads



e.g., Libtask (swtch.com/libtask)

```c
void taskmain(int argc, char **argv) {
    for (int i=0; i<3; i++) {
        /* specify task fn, arg, stack size */
        taskcreate(task_fn, (void *)i, 32768);
    }
}

int glob = 0;

void task_fn(void *num) {
    for (int i=0; i<5; i++) {
        printf("Task %d: glob = %d\n", (int)num, glob);
        for (int j=0; j<1000; j++) {
            glob += 1;
        }
        taskyield(); /* give up CPU */
    }
}
```

```
Task 0: glob = 0
Task 1: glob = 1000
Task 2: glob = 2000
Task 0: glob = 3000
Task 1: glob = 4000
Task 2: glob = 5000
Task 0: glob = 6000
Task 1: glob = 7000
Task 2: glob = 8000
Task 0: glob = 9000
Task 1: glob = 10000
Task 2: glob = 11000
Task 0: glob = 12000
Task 1: glob = 13000
Task 2: glob = 14000
```

```c
Task **alltask;

taskcreate(void (*fn)(void*), void *arg, uint stack)
{
  Task *t;
  t = taskalloc(fn, arg, stack);
  taskcount++;
  id = t->id;
  t->alltaskslot = nalltask;
  alltask[nalltask++] = t;
  ...
}

taskyield(void)
{
  taskswitch();
  ...
}

taskswitch(void)
{
  contextswitch(&taskrunning->context, &taskschedcontext);
}
```

```c
static Task*
taskalloc(void (*fn)(void*), void *arg, uint stack)
{
   Task *t;

   /* allocate the task and stack together */
   t = malloc(sizeof *t+stack);
   memset(t, 0, sizeof *t);
   t->stk = (uchar*)(t+1);
   t->stksize = stack;
   t->id = ++taskidgen;
   t->startfn = fn;
   t->startarg = arg;

   /* do a reasonable initialization */
   memset(&t->context.uc, 0, sizeof t->context.uc);
   ...

   return t;
}
```

```c
void contextswitch(Context *from, Context *to) {
    swapcontext(&from->uc, &to->uc);
    ...
}

int swapcontext(ucontext_t *oucp, ucontext_t *ucp) {
    if(getcontext(oucp) == 0)
        setcontext(ucp);
    return 0;
}

struct ucontext {
    mcontext_t  uc_mcontext;
    ...
};

struct mcontext {
    ...
    int  mc_ebp;
    ...
    int  mc_ecx;
    int  mc_eax;
    ...
    int  mc_eip;
    int  mc_cs;
    int  mc_eflags;
    int  mc_esp;
    ...
};
```

```
#define setcontext(u)  SET(&(u)->uc_mcontext)
#define getcontext(u)  GET(&(u)->uc_mcontext)


GET:
    movl   4(%esp), %eax   /* %eax=arg */
    ...
    movl   %ebp, 28(%eax)
    ...
    movl   $1, 48(%eax)    /* %eax */
    movl   (%esp), %ecx    /* %eip */
    movl   %ecx, 60(%eax)
    leal   4(%esp), %ecx   /* %esp */
    movl   %ecx, 72(%eax)
    movl   $0, %eax
    ret

SET:
    movl   4(%esp), %eax   /* %eax=arg */
    ...
    movl   28(%eax), %ebp
    ...
    movl   72(%eax), %esp
    pushl  60(%eax)        /* new %eip */
    movl   48(%eax), %eax
    ret
```

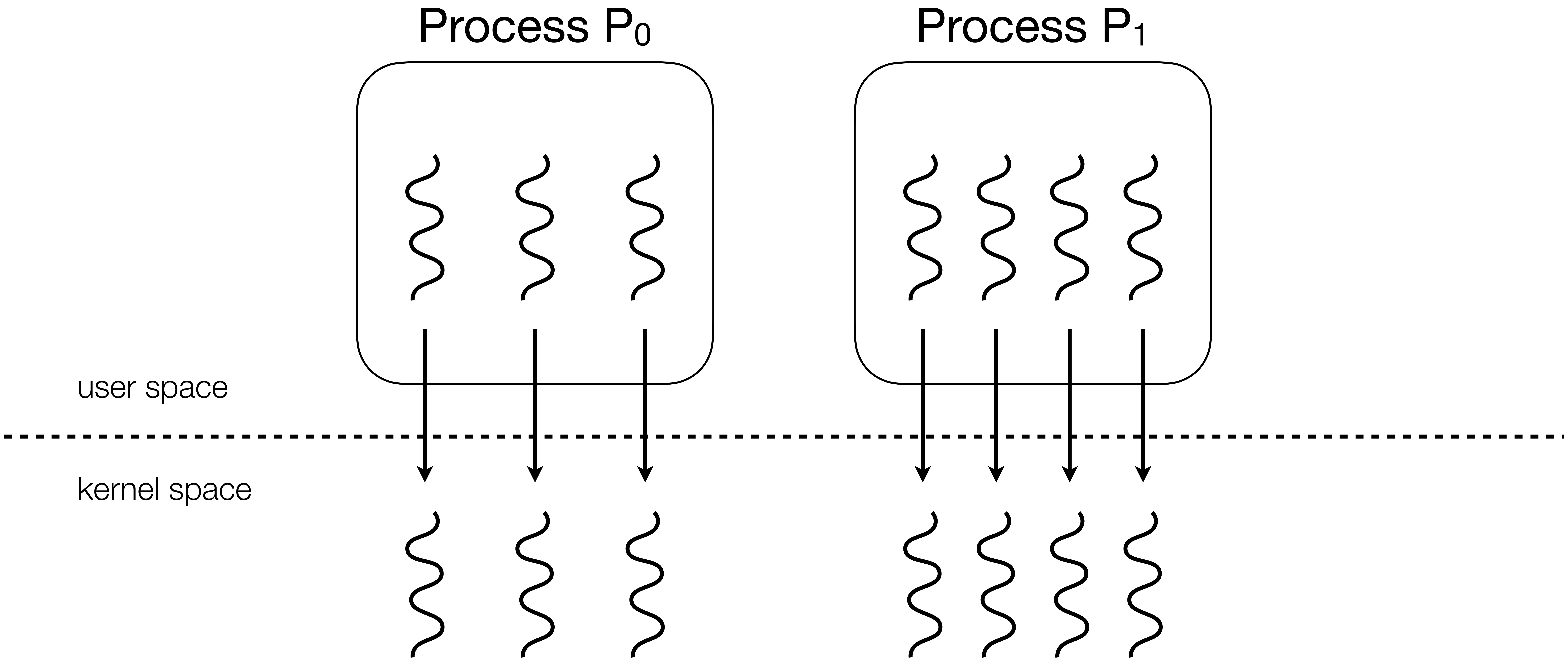
User-level threads pros/cons

- Pros
  - Lightweight implementation
    - No kernel overhead
  - Context switching is fast
    - No need to switch to kernel
  - Portable (OS-independent)
- Cons
  - Reinvents the wheel (scheduler)
  - Cannot run on multiple CPUs (no parallelism)
    - Only one scheduling entity known to kernel
  - Multithreaded task is treated the same as a single-threaded task

Kernel-level (aka native) threads

- Kernel is aware of all threads in each process
  - TCBs stored in kernel space
- Thread creation and scheduling carried out by kernel
  - Context switch between threads in the same process is cheaper (why?) than inter-process context switch, but still requires interrupt/trap

1:1 mapping of user→kernel threads



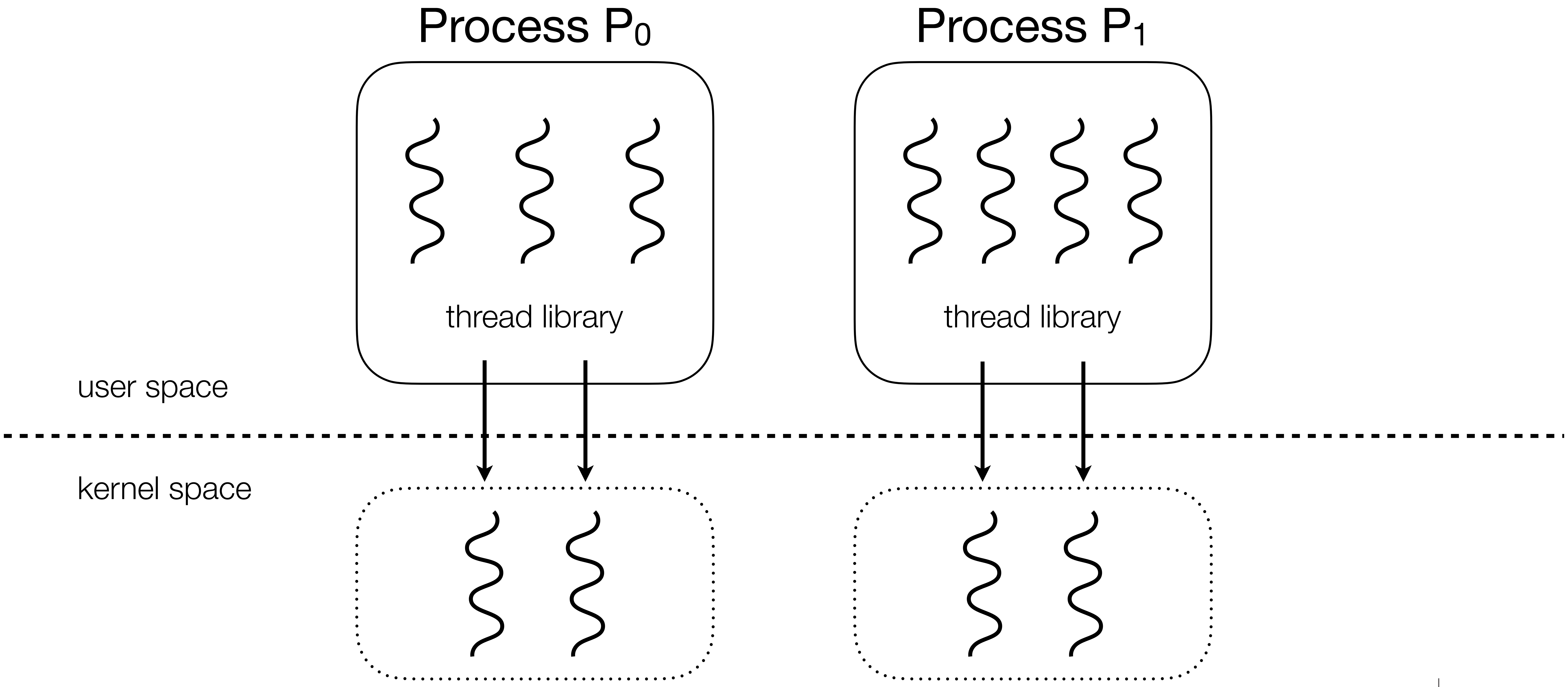
Kernel-level threads pros/cons

- Pros
  - Reuses scheduler for threads
  - Support for intra-process thread-level parallelism
    - Can take advantage of multiple CPUs
- Cons
  - Threads are “heavyweight” system entities
    - Much more expensive to create and maintain

Compromise: hybrid model

- Kernel provides a limited number of scheduling entities to each process; user code is responsible for running a user thread in each entity
  - Supports fast thread context switches and parallel execution
  - Limits total thread burden on system
  - At cost of increased complexity and user/kernel coupling

M:N mapping of user→kernel threads



Threading APIs

- Threading APIs support thread creation, management, and coordination
  - May be language/library/runtime/OS-specific
  - Many modern libraries support user-level threads
- Most popular Unix low-level threading API = POSIX threads, “pthreads”
- OpenMP is a more abstract threading API for exploiting parallelism

POSIX threads (pthreads)

- C language threading API — 100+ functions in 4 categories
  - Thread management
  - Mutexes
  - Condition variables
  - Synchronization

more on these later!

- API doesn't specify a user- or kernel- level thread implementation
  - Most modern Unix implementations support 1:1 or M:N threading

e.g., pthreads thread mgmt API

```
/* thread creation */
int pthread_create (pthread_t *tid,
                    const pthread_attr_t *attr,
                    void *(*thread_fn)(void *),
                    void *arg );

/* wait for termination; thread "reaping" */
int pthread_join (pthread_t tid,
                  void **result_ptr );

/* terminates calling thread */
int pthread_exit (void *value_ptr );
```

```c
int counter = 0;

void *inc(void *num) {
    for (int i=0; i<10000; i++) {
        counter += 1;
    }
    printf("Thread %ld counter = %d\n",
           pthread_self(), counter);
    pthread_exit(NULL);
}

int main() {
    pthread_t tid;
    for (int i=0; i<5; i++){
        pthread_create(&tid, NULL, inc, NULL);
        printf("Created thread %ld\n", tid);
    }
    pthread_exit(NULL); // terminate main thread
    return 0; // never get here!
}
```

Run 1:

```
Created thread 139859278001920
Thread 139859278001920 counter = 10000
Created thread 139859269609216
Thread 139859269609216 counter = 20000
Created thread 139859261216512
Thread 139859261216512 counter = 30000
Created thread 139859252713216
Created thread 139859244320512
Thread 139859252713216 counter = 40000
Thread 139859244320512 counter = 50000
```

Run 2:

(?!?)

```
Created thread 139949404641024
Created thread 139949396248320
Created thread 139949387855616
Thread 139949396248320 counter = 20035
Created thread 139949379462912
Thread 139949404641024 counter = 10000
Created thread 139949371070208
Thread 139949387855616 counter = 20833
Thread 139949379462912 counter = 28523
Thread 139949371070208 counter = 34961
```