Instituto de Matemática e Estatística

USP

MAC422
Sistemas Operacionais

Prof. Marcel P. Jackowski

Aula #8

Threads e sincronização

Referências: Sistemas Operacionais Modernos (3a. Ed.), aulas antigas de MAC-422 (IME-USP), W4118 (Columbia University)

fork()

```
// Create a new process copying p as the parent.
// Sets up stack to return as if from system call.
// Caller must set state of returned proc to RUNNABLE.
int
fork(void)
{
  int i, pid;
  struct proc *np;
  // Allocate process.
  if((np = allocproc()) == 0)
    return -1;

  // Copy process state from p.
 if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
    kfree(np->kstack);
  np->kstack = 0;
  np->state = UNUSED;
  return -1;
}
np->sz = proc->sz;
np->parent = proc;
*np->tf = *proc->tf;

// Clear %eax so that fork returns 0
// in the child.
np->tf->eax = 0;

for(i = 0; i < NOFILE; i++)
  if(proc->ofile[i])
    np->ofile[i] = filedup(proc->ofile[i]);
np->cwd = idup(proc->cwd);

  pid = np->pid;
  np->state = RUNNABLE;
  safestrcpy(np->name, proc->name, sizeof
(proc->name));
  return pid;
}
```

exit()

```
// Exit the current process.  Does not return.
// An exited process remains in the zombie state
// until its parent calls wait() to find out it exited.
void
exit(void)
{
  struct proc *p;
  int fd;

  if(proc == initproc)
    panic("init exiting");

  // Close all open files.
  for(fd = 0; fd < NOFILE; fd++){
    if(proc->ofile[fd]){
      fileclose(proc->ofile[fd]);
      proc->ofile[fd] = 0;
    }
  }

  iput(proc->cwd);
  proc->cwd = 0;

  acquire(&ptable.lock);

  // Parent might be sleeping in wait().
  wakeup1(proc->parent);

  // Pass abandoned children to init.
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == proc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }

  // Jump into the scheduler, never to return.
  proc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

wait()

```
// Wait for a child process to exit and return its pid.
// Return -1 if this process has no children.
int
wait(void)
{
  struct proc *p; int havekids, pid;
  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for zombie children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
      if(p->parent != proc)
       continue;
      havekids = 1;
      if(p->state == ZOMBIE) { // Found one.
        pid = p->pid;
        kfree(p->kstack); p->kstack = 0; freevm(p->pgdir);
        p->state = UNUSED; p->pid = 0; p->parent = 0; p->name[0] = 0; p->killed = 0;
        release(&ptable.lock);
        return pid;
      }
    }

    // No point waiting if we don't have any children.
    if(!havekids || proc->killed){
      release(&ptable.lock); return -1;
    }

    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
    sleep(proc, &ptable.lock);  //DOC: wait-sleep
  }
```

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX 10
void print(int *v)
{
  int i;

  for(i=0; i<MAX; i++)
    printf("%d ",v[i]);

  printf("\n");
}

int main(void)
{
int i, x;
int *y = (int *) malloc (MAX * sizeof(int));

for(i=0;i<10;i++) y[i] = i;

pid_t pid = fork();

if (pid == 0) {

  printf("I am in the child process, y=%x\n",y);

  sleep(2);

  /* print vector */
  print(y);

  free(y);
```

```c
} else if (pid > 0) {
   /*
      If fork() returns a positive number,
      we are in the parent process and pid is the
      child process id.
   */

   printf("I am in the parent process, y=%x\n",y);

   y[9] = 0;

   /* print vector */
   print(y);

   free(y);

 } else {
   fprintf(stderr, "Error, can't fork!\n");

   exit(1);
 }

 exit(0);
}
```

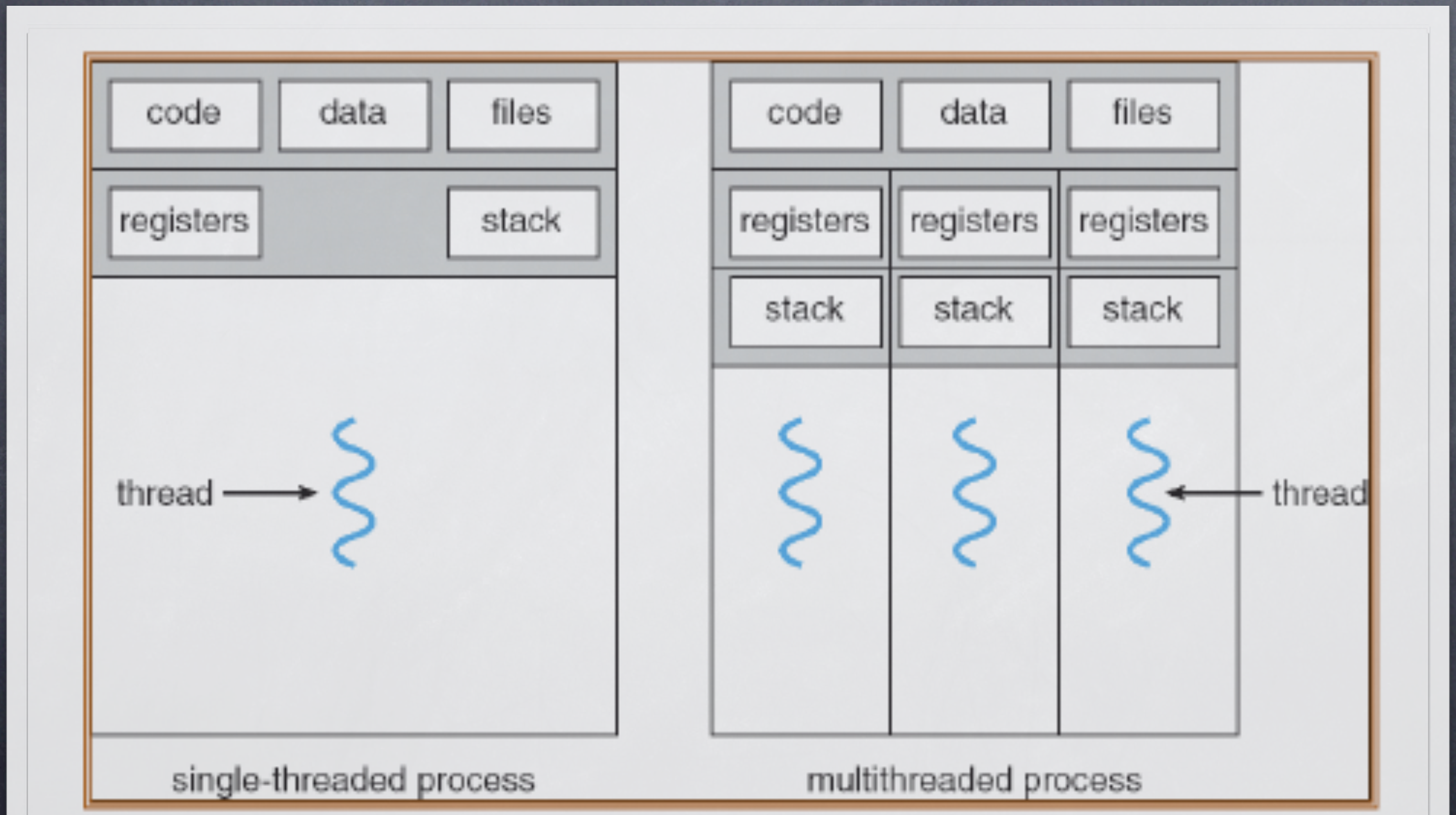
O que acontece com o espaço alocado por malloc() ?

Threads

- Threads: separate streams of executions that share an address space
  - Allows one process to have multiple point of executions, can potentially use multiple CPUs

- Thread control block (TCB)
  - Program counter (EIP on x86)
  - Other registers
  - Stack

Single-threaded e multithreaded



Por quê threads ?

- Express concurrency
  - Web server (multiple requests), Browser (GUI + network I/O + rendering), …

```
for(;;) {
    struct request *req = get_request();
    create_thread(process_request, req);
}
```

- Efficient communication
  - Using a separate process for each task can be heavyweight

Threads vs. Processos

- A thread has no data segment or heap
- A thread cannot live on its own, it must live within a process
- There can be more than one thread in a process, the first thread calls main() & has the process's stack
- Inexpensive creation
- Inexpensive context switching
- Efficient communication
- If a thread dies, its stack is reclaimed

- A process has code/data/heap & other segments
- A process has at least one thread
- Threads within a process share code/data/heap, share I/O, but each has its own stack & registers
- Expensive creation
- Expensive context switching
- Interprocess communication can be expressive
- If a process dies, its resources are reclaimed & all threads die

Utilizando threads

- Through thread library
  - E.g. pthread, Win32 thread
- Common operations
  - create/terminate
  - suspend/resume
  - priorities and scheduling
  - synchronization

Pthreads

- int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg);
  - Create a new thread to run start_routine on arg
  - thread holds the new thread's id
  - Can be customized via attr

- int pthread_join(pthread_t thread, void **value_ptr);
  - Wait for thread termination, and retrieve return value in value_ptr

- void pthread_exit(void *value_ptr);
  - Terminates the calling thread, and returns value_ptr to threads waiting in pthread_join

Exemplo de criação de threads

```
void* thread_fn(void *arg)
{
    int id = (int)arg;
    printf("thread %d runs\n", id);
    return NULL;
}

int main()
{
    pthread_t t1, t2;
    pthread_create(&t1, NULL, thread_fn, (void*)1);
    pthread_create(&t2, NULL, thread_fn, (void*)2);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}
```

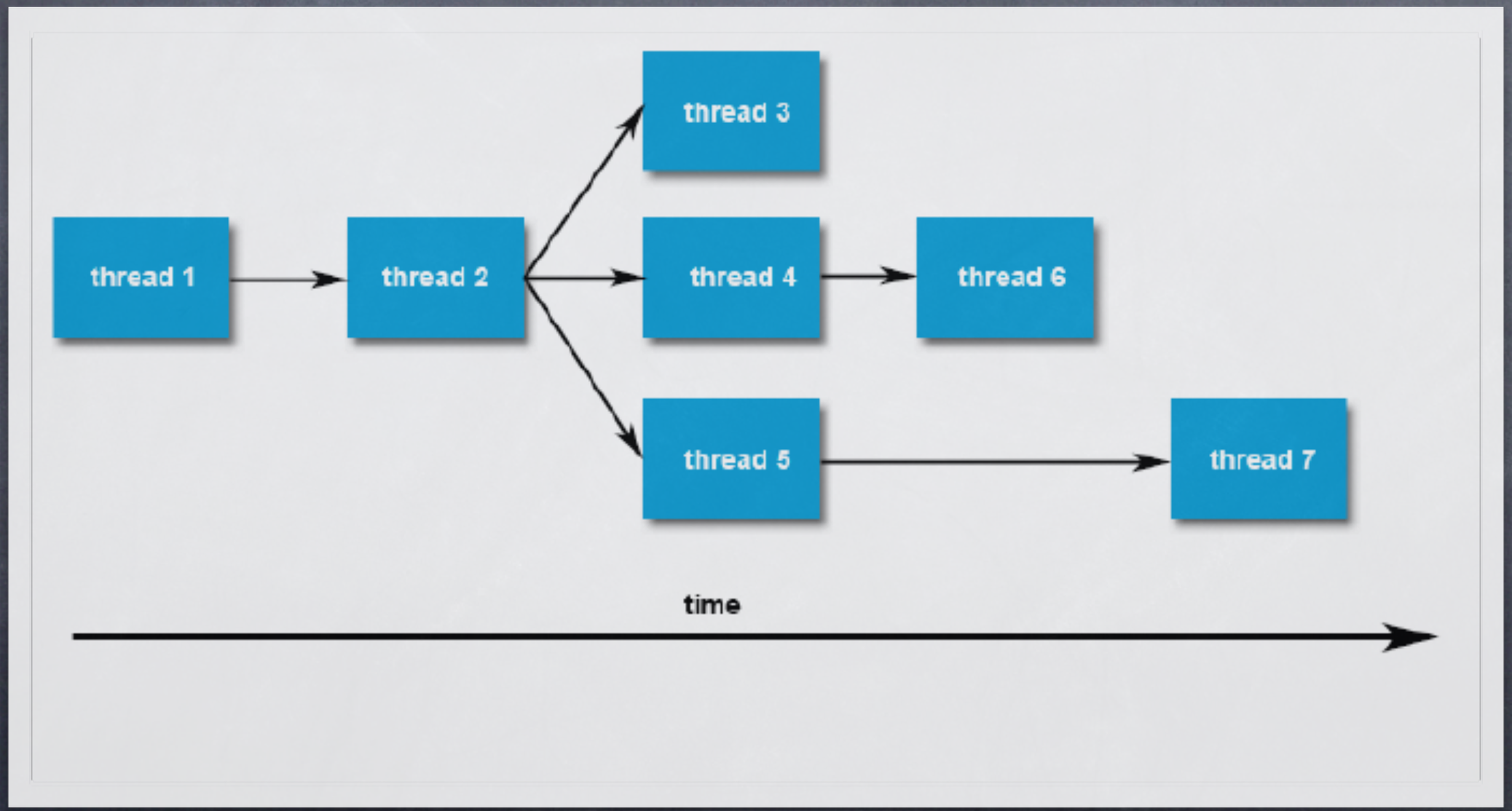
```
$ gcc –o threads threads.c –Wall –lpthread
$ threads
thread 1 runs
thread 2 runs
```

One way to view threads: function calls, except caller doesn't wait for callee; instead, both run concurrently

Exemplo de pthreads

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.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; return code from pthread_create() is %d\n", rc);
      exit(-1);
      }
    }
  pthread_exit(NULL);
}
```

Threads podem criar outros threads

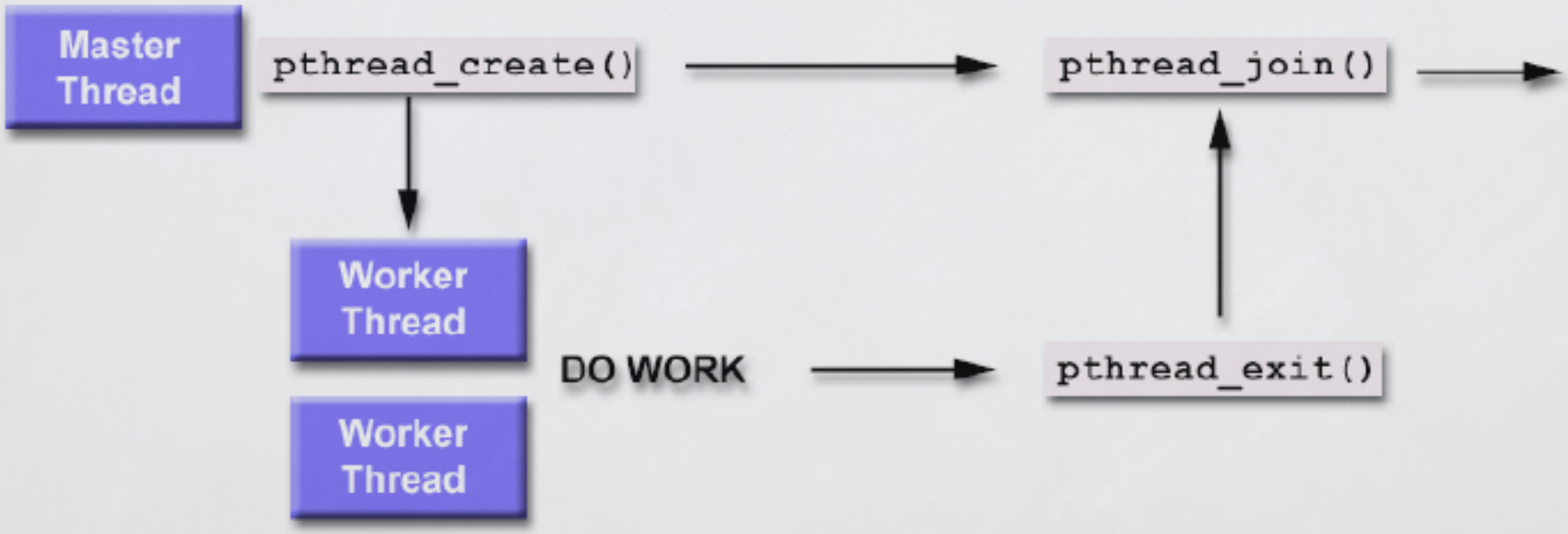


Quem é escalonado primeiro ?

Terminando threads

- No término de sua rotina de execução
  - A thread chama pthread_exit()
  - A thread é cancelada por outra com o comando pthread_cancel()
  - O processo inteiro é cancelado através de uma chamada a exit() ou exec().

Sincronizando o trabalho



```c
int main(int argc, char *argv[])
{
   pthread_t thread[NUM_THREADS];
   pthread_attr_t attr;
   int rc, t;
   void *status;

   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
   for(t=0;t<NUM_THREADS;t++)
   {
      printf("Creating thread %d\n", t);
      rc = pthread_create(&thread[t], &attr, BusyWork, NULL);
      if (rc) {
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1); }
   }
   pthread_attr_destroy(&attr);
   for(t=0;t<NUM_THREADS;t++)
   {
      rc = pthread_join(thread[t], &status);
      if (rc) {
         printf("ERROR return code from pthread_join() is %d\n", rc);
         exit(-1);}
      printf("Completed join with thread %d status= %ld\n",t,(long)status);
   }
   pthread_exit(NULL);
}
```

Continuação

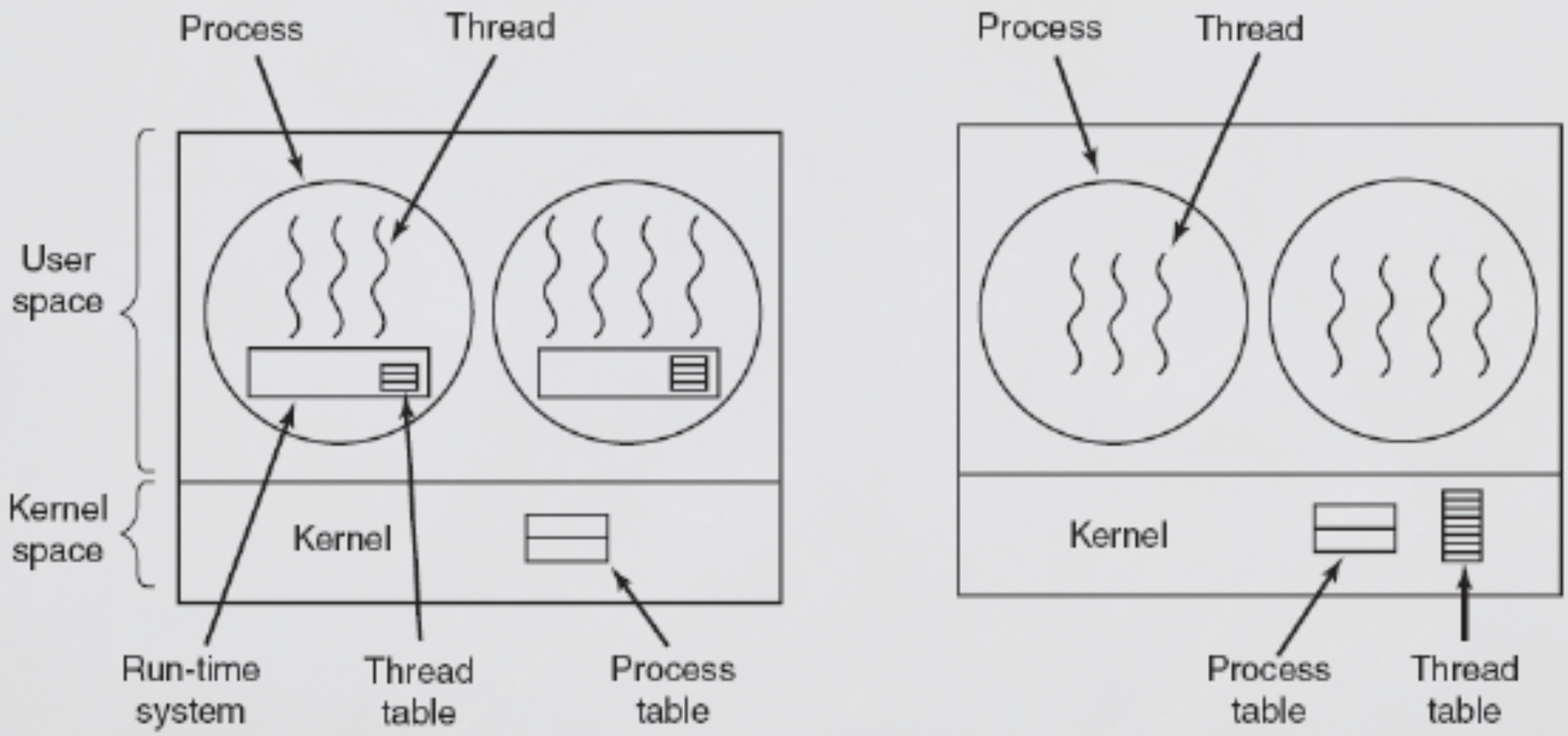
```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS3

void *BusyWork(void *null)
{
   int i;
   double result=0.0;
   for (i=0; i<1000000; i++)
   {
     result = result + (double)random();
   }
   printf("Thread result = %e\n",result);
   pthread_exit((void *) 0);
}
```

Modelos de multithreading

- Where to support threads?
- User threads: thread management done by user-level threads library; kernel knows nothing
- Kernel threads: threads directly supported by the kernel
  - Virtually all modern OS support kernel threads

User vs. kernel threads



User threads vs. kernel threads

- Pros: fast, no system call for creation, context switch
- Cons: kernel doesn't know → one thread blocks, all threads in the process blocks

- Cons: slow, kernel does creation, scheduling, etc
- Pros: kernel knows → one thread blocks, schedule another

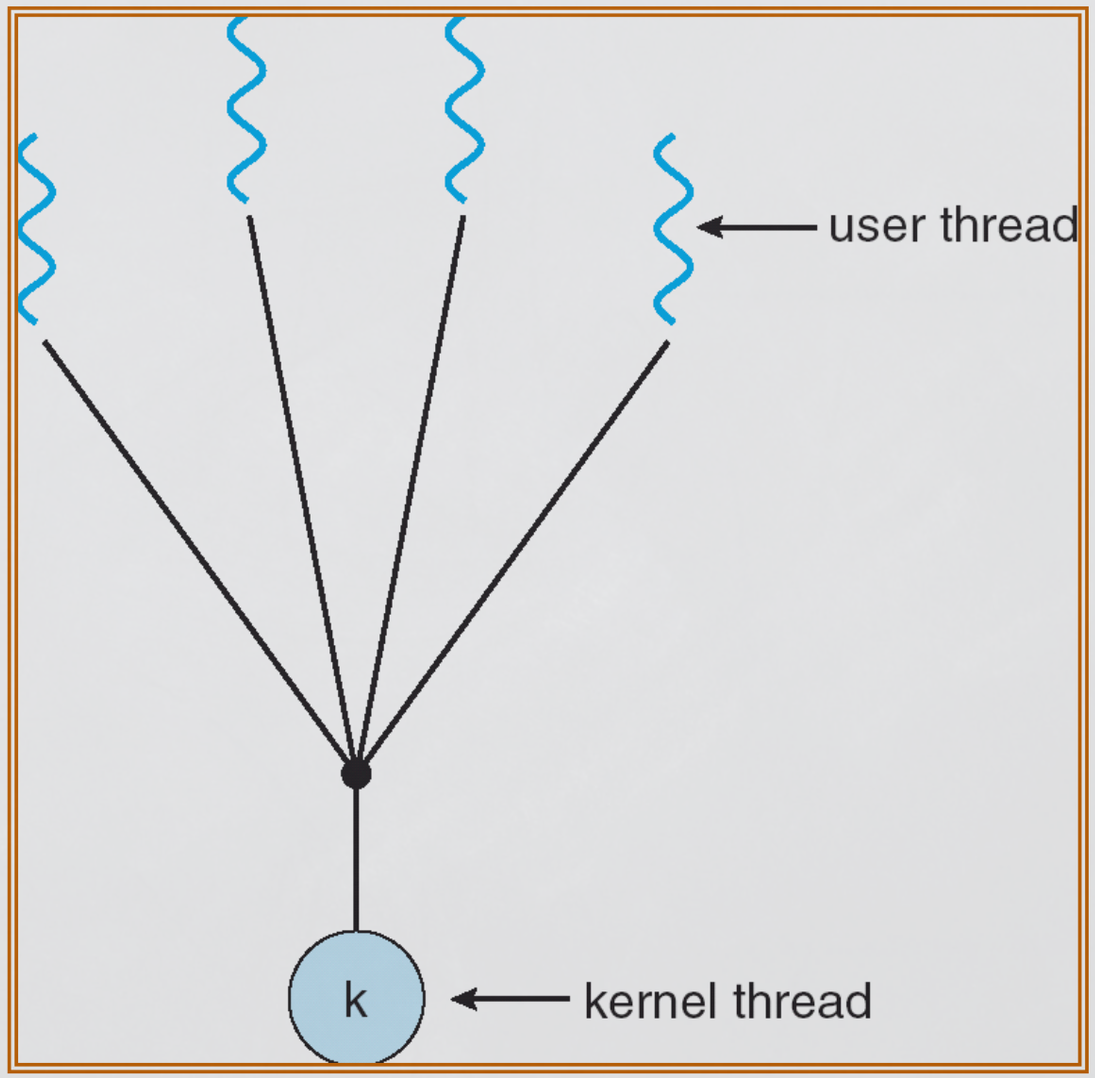
No free lunch!

Implementação de threads

- A thread library must map user threads to kernel threads
- Big picture:
  - kernel thread: physical concurrency, how many cores?
  - User thread: application concurrency, how many tasks?
- Different mappings exist, representing different tradeoffs
  - Many-to-One: many user threads map to one kernel thread, i.e. kernel sees a single process
  - One-to-One: one user thread maps to one kernel thread
  - Many-to-Many: many user threads map to many kernel threads

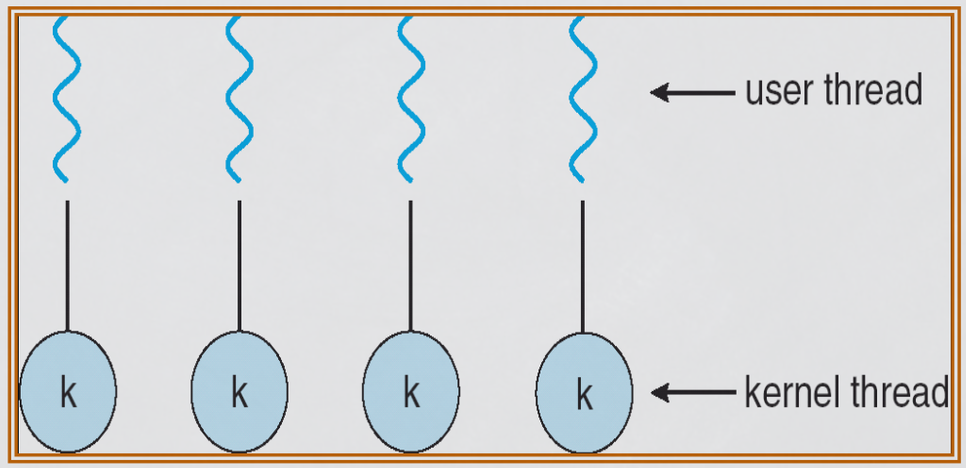
"Many-to-one"

- Many user-level threads map to one kernel thread
- Pros
  - Fast: no system calls required
  - Portable: few system dependencies
- Cons
  - No parallel execution of threads
    - All thread block when one waits for I/O



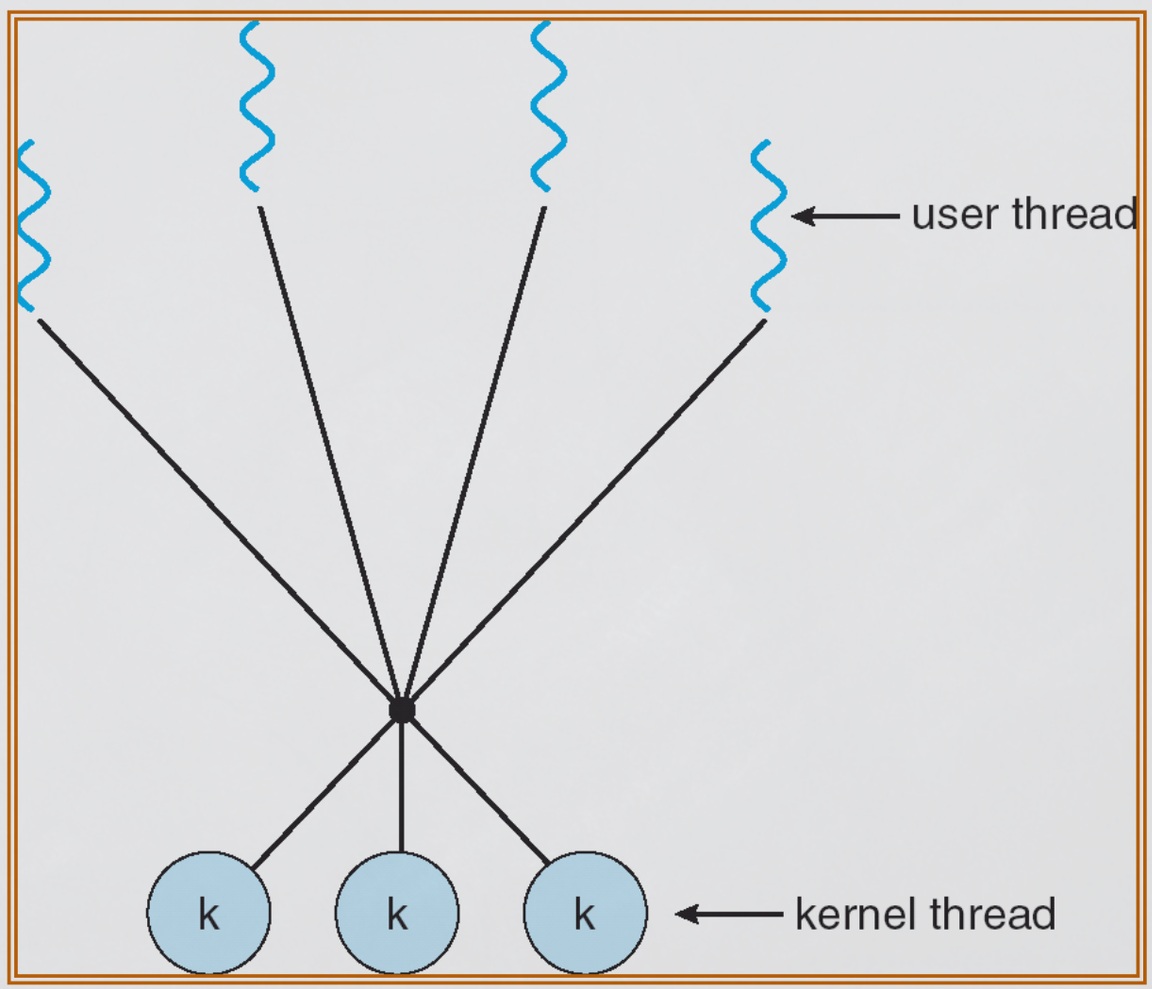
"One-to-one"

- One user-level thread maps to one kernel thread
- Pros: more concurrency
  - When one blocks, others can run
  - Better multicore or multiprocessor performance
- Cons: expensive
  - Thread operations involve kernel
  - Thread need kernel resources



"Many-to-many"

- Many user-level threads map to many kernel threads (U >= K)
  - Supported some versons of BSD, and Windows
- Pros: flexible
  - OS creates kernel threads for physical concurrency
  - Applications creates user threads for application concurrency
- Cons: complex
  - Most programs use 1:1 mapping anyway



Threads em Linux

- Unidade de escalonamento: “tasks”
- Criação através do *syscall* “clone()”
- Utilização do modelo “kernel threads”
  - One-to-one
- Exemplos de informações que são compartilhadas entre threads:
  - Espaço em memória
  - Descritores de arquivos
  - Gerenciadores de sinais

Implementação em Windows™

- Processos não são escalonáveis.
- Cada processo possui inicialmente 1 thread.
  - Threads são escalonáveis
- Novos threads são criados através da *syscall*
  - CreateThread()
  - Passa-se o endereço da função ou código a ser executado.
- Fibras ("*fibers*")

Threads em Java

- Podemos usar herança e criamos uma classe que estende a classe Thread
- Devemos redefinir o método run():

```
class Tarefa1 extends Thread {
  public void run() {
    for(int i=0; i<1000; i++)
      System.out.println("Usando herança");
  }
}

Thread threadComHeranca = new Tarefa1();
threadComHeranca.start();
```

Threads em Java

- A interface Runnable nos obriga a implementar o método run():

```
class Tarefa2 implements Runnable {
  public void run() {
    for(int i=0; i<1000; i++)
      System.out.println(“Usando Runnable”);
  }
}

Thread threadComRunnable = new Thread(new Tarefa2());
threadComRunnable.start();
```

"Thread pool"

- Problem:
  - Creating a thread for each request: costly
    - And, the created thread exits after serving a request
  - More user request ➔ More threads, server overload

- Solution: thread pool
  - Pre-create a number of threads waiting for work
  - Wake up thread to serve user request --- faster than thread creation
  - When request done, don't exit --- go back to pool
  - Limits the max number of threads

Transação bancária

```
int balance = 1000;

int main()
{
    pthread_t t1, t2;
    pthread_create(&t1, NULL, deposit, (void*)1);
    pthread_create(&t2, NULL, withdraw, (void*)2);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("all done: balance = %d\n", balance);
    return 0;
}
```

```
void* deposit(void *arg)
{
    int i;
    for(i=0; i<1e7; ++i)
        ++balance;
}

void* withdraw(void *arg)
{
    int i;
    for(i=0; i<1e7; ++i)
        --balance;
}
```

Possíveis resultados

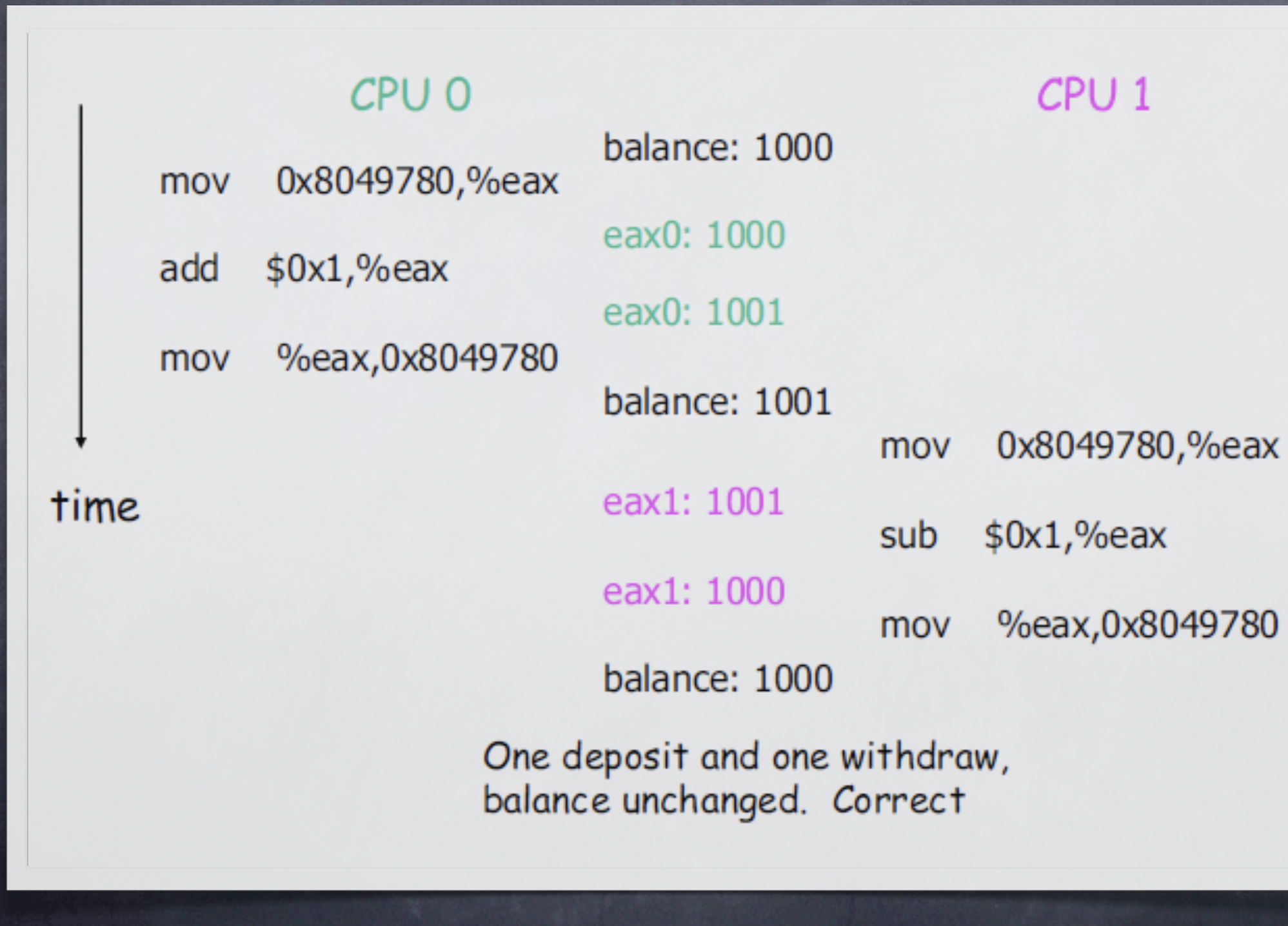
```
$ gcc –Wall –lpthread –o bank bank.c
$ bank
all done: balance = 0
$ bank
all done: balance = 140020
$ bank
all done: balance = -94304
$ bank
all done: balance = -191009
```

Por quê ?

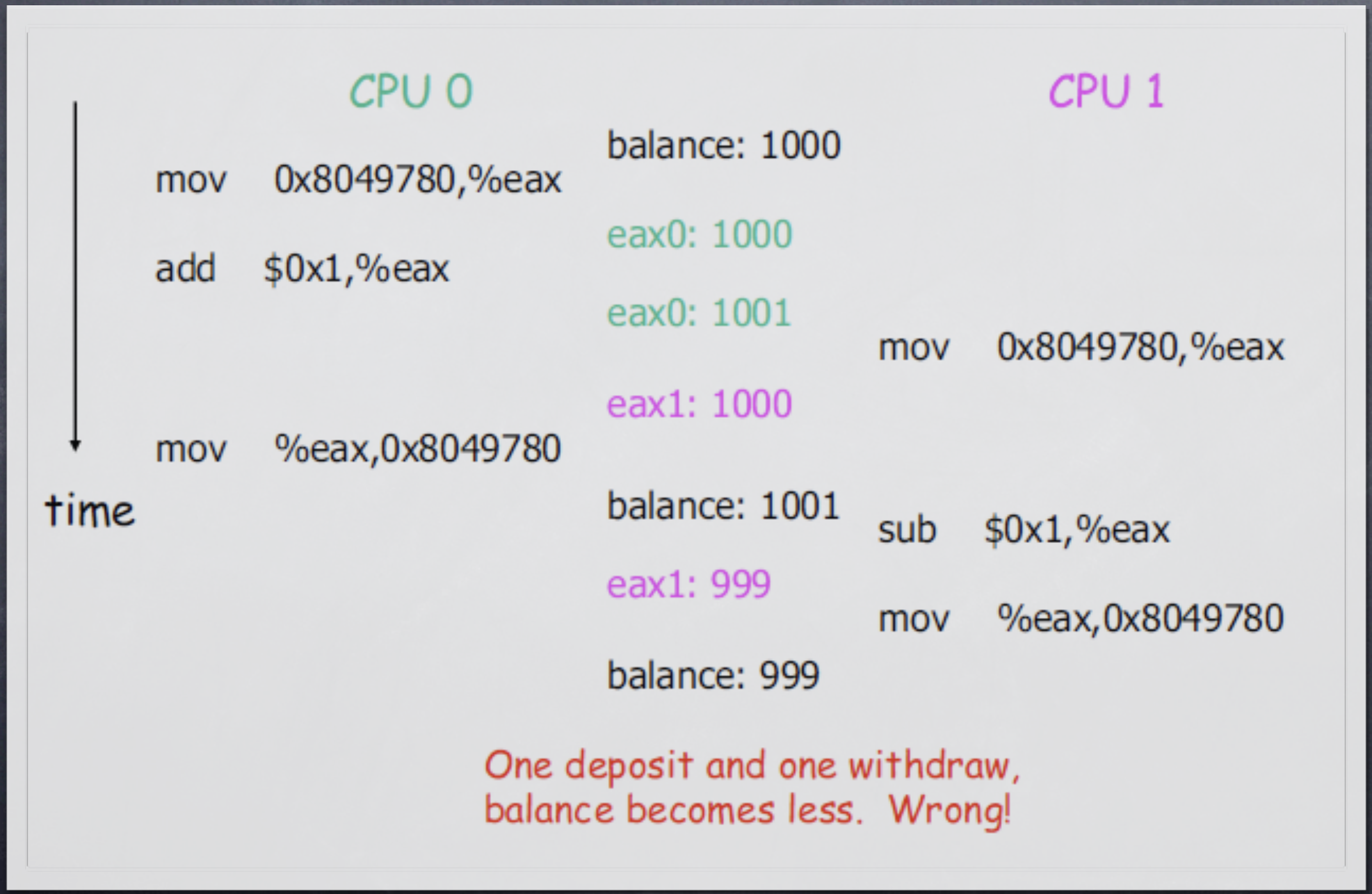
Olhando mais a fundo...

```
$ objdump –d bank
...
08048464 <deposit>:
... // ++ balance
8048473: a1 80 97 04 08    mov 0x8049780,%eax
8048478: 83 c0 01          add $0x1,%eax
804847b: a3 80 97 04 08    mov %eax,0x8049780
...
0804849b <withdraw>:
… // -- balance
80484aa: a1 80 97 04 08    mov 0x8049780,%eax
80484af: 83 e8 01          sub $0x1,%eax
80484b2: a3 80 97 04 08    mov %eax,0x8049780
```

Possível escalonamento



Outro escalonamento



"Race condition"

- Definition: a timing dependent error involving shared state

- Can be very bad
  - "non-deterministic:" don't know what the output will be, and it is likely to be different across runs
  - Hard to detect: too many possible schedules
  - Hard to debug: "heisenbug," debugging changes timing so hides bugs (vs "bohr bug"

Como evitar as "race conditions" ?

- Atomic operations: no other instructions can be interleaved, executed "as a unit" "all or none", guaranteed by hardware

- A possible solution: create a super instruction that does what we want atomically
  - add $0x1, 0x8049780

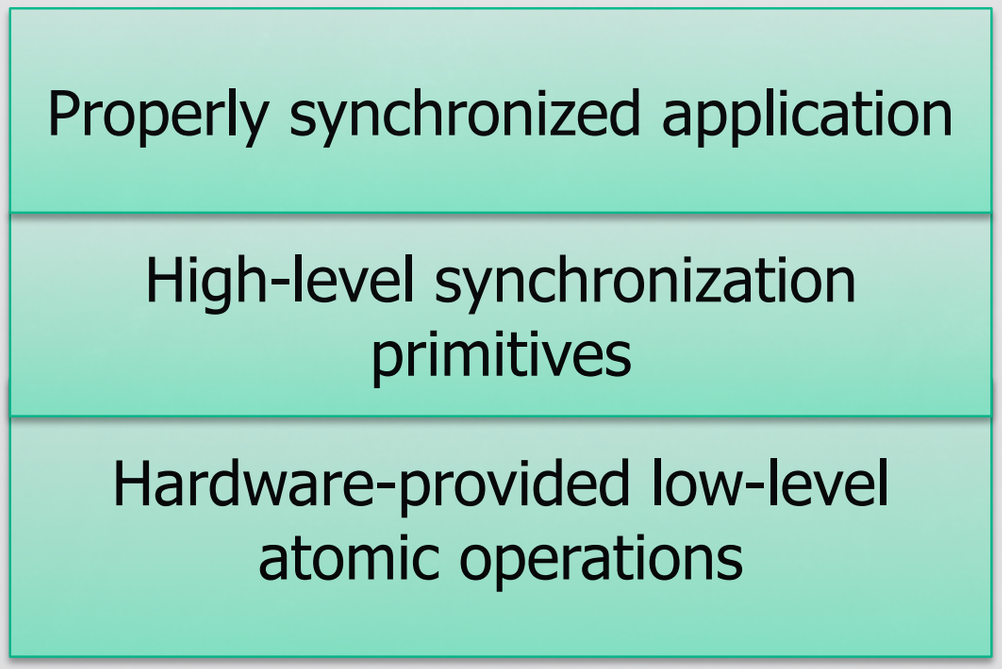
- Problem
  - Can't anticipate every possible way we want atomicity
  - Increases hardware complexity, slows down other instructions

```
// ++ balance
mov    0x8049780,%eax
add    $0x1,%eax
mov    %eax,0x8049780
...

// -- balance
mov    0x8049780,%eax
sub    $0x1,%eax
mov    %eax,0x8049780
...
```

Sincronização por camadas

- Hardware provides simple low-level atomic operations, upon which we can build high-level, synchronization primitives, upon which we can implement critical sections and build correct multi-threaded/multi-process programs



Primitivas de sincronização

- Low-level atomic operations
  - On uniprocessor, disable/enable interrupt
  - On x86, aligned load and store of words
  - Special instructions:
    - test-and-set (TSL), compare-and-swap (XCHG)

- High-level synchronization primitives
  - Lock
  - Semaphore
  - Monitor

Tarefa de Casa

- Leitura sobre o xv6:
  - Veja como exec() funciona no código fonte
    - sysfile.c
    - exec.c
- Documentação sobre o xv6:
  - exec.pdf
  - lock.pdf