EC 440 – Introduction to Operating Systems

Manuel Egele

Department of Electrical & Computer Engineering

Boston University

The Process Concept

- The OS creates number of virtual computers
- Execution of a program on one of these virtual computers is called a *sequential process*
- The virtual computer gives the illusion to each process that it is running on a dedicated CPU with a dedicated memory
- The actual CPU is switched back and forth among the processes (multiprogramming with time-sharing)
- Process memory is managed so that all the needed portions are present in the actual memory
- The virtual computer is the execution environment, the process is the executor, and the program being executed determines the process behavior

Programs vs. Processes

Program

- Static object existing in a file
- A sequence of instruction
- Static existence in space & time
- Same program can be executed by different processes

Process

- Dynamic object – program in execution
- A sequence of instruction executions
- Exists in limited span of time
- Same process may execute different program

```
main() {
    int i, prod = 1;
    for (i=0 ; i < 100; i++)
        prod = prod * i;
}
```

```
prod = prod*i;
```

Process executes it 100 times

Process Life Cycle

A process can be created

- During OS initialization
  - “init” process in UNIX
- By another process
  - `fork()`, or `NtCreateProcess()`

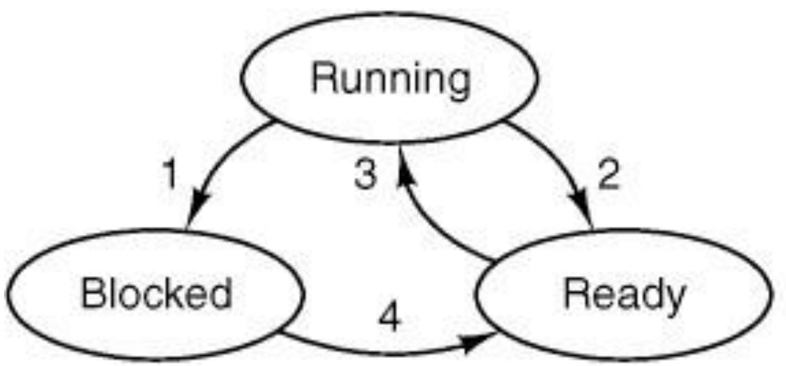
A process can be terminated

- By itself
  - `exit()`, or `ExitProcess()`
- Because of an error
  - e.g., segmentation fault
- By another process
  - `kill()`, `TerminateProcess()`

Process States

Process states

- Running (using the CPU)
- Ready (waiting for the CPU)
- Blocked (waiting for a resource to become available)



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Process States – Blocked

What can cause processes to block?

– Waiting for input

– Explicit sleep

Inputs can be, for example:

– Network I/O

– Disk

– Waiting for the user to click something

– etc.

Processes

Process hierarchy (Unix)

– each process (except init) has exactly one parent
– each process can have many children

Parent must collect status of child processes

– otherwise … children become *zombie* processes
– what happens when parent dies first?

How is signal delivery handled

– i.e., do children receive signals of parents? – Yes.

Processes Windows

- Windows does keep track of parent/child relationships
  - Relationships don't affect anything though
- When a process is created (`NtCreateProcess`) parent gets a handle – a token – so it can control the child

```
NTAPI NtCreateProcessEx(
    OUT PHandle ProcessHandle
    …);
```

- Handles can be passed to other processes, and privileged processes can obtain handles to other processes (e.g., via `OpenProcess`)

Process Implementation

The OS maintains a *process table* with an entry for each process, called *Process Control Block* (PCB)

The PCB contains:

- Process ID, User ID, Group ID
- Process state (Running, Ready, Blocked)
- Registers (Program counter, PSW, Stack pointer, etc)
- ***Pointers to*** memory segments (Stack, Heap, Data, Text)
- Priority/Scheduling parameters
- Accounting information
- Signal management functions
- Open file tables
- Working directory

PCB In Linux – task_struct

```
struct task_struct {
    volatile long state;  /* -1 unrunnable, 0 runnable, >0 stopped */
    /* task state */
    int exit_state;
    pid_t pid;
    /* Canary value for the -fstack-protector gcc feature */
    unsigned long stack_canary;
    struct task_struct *parent; /* recipient of SIGCHLD, wait4() reports */
    struct timespec start_time;        /* monotonic time */
    char comm[TASK_COMM_LEN]; /* executable name excluding path */
    /* CPU-specific state of this task */
    struct thread_struct thread;
    /* signal handlers */
    struct signal_struct *signal;
    sigset_t blocked, real_blocked;
    ... 300 lines ...
}
```

Process Implementation

In Minix, different pieces of information about a process are stored in different parts of the OS

Kernel

- register values (PC, stack pointer, …)
- scheduling information

Process management

- memory information (pointers to text, data, bss segment)
- IDs (UID, GID, …)

File management

- working directory
- umask
- file table

Threads

A process is a way to

- Group resources (memory, open files, ...)
- Perform the execution of a program: a thread of execution (code, program counter, registers, stack)

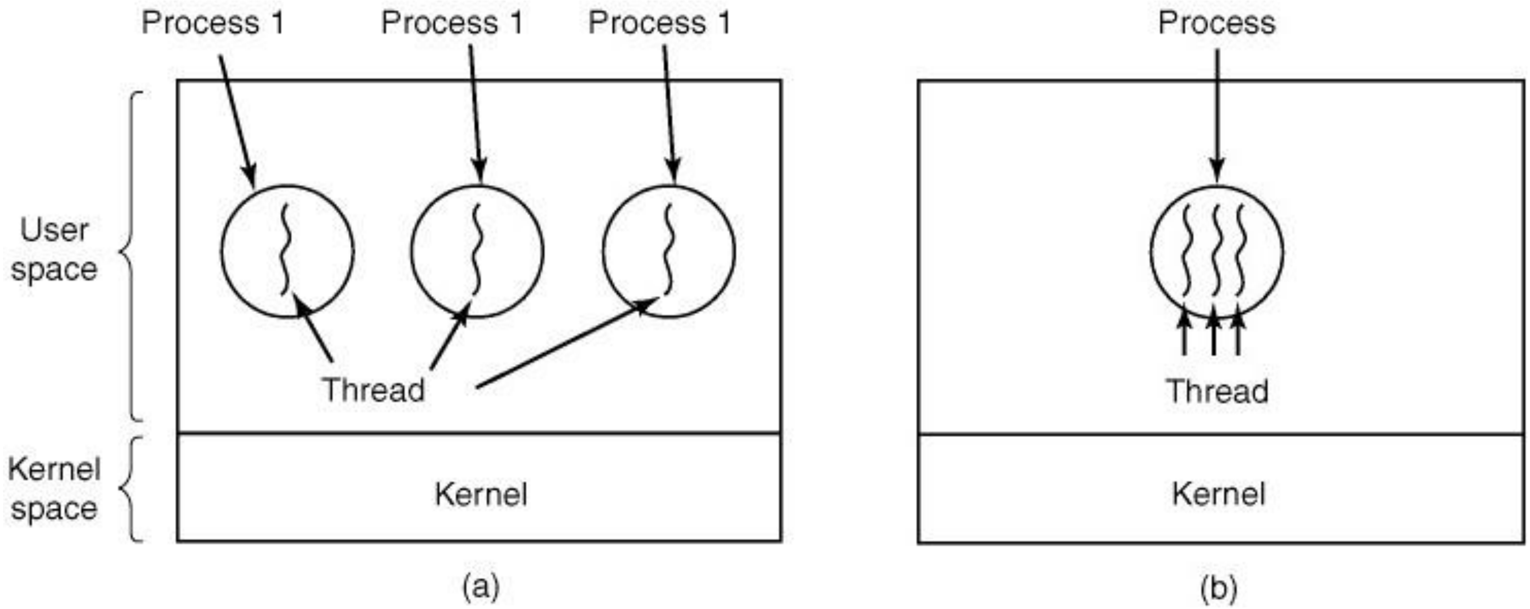
Multiple threads of execution can run in the same process environment

Multiple threads (in the same process) share

- Common address space (shared memory)
- Open files
- Process, user, and group IDs

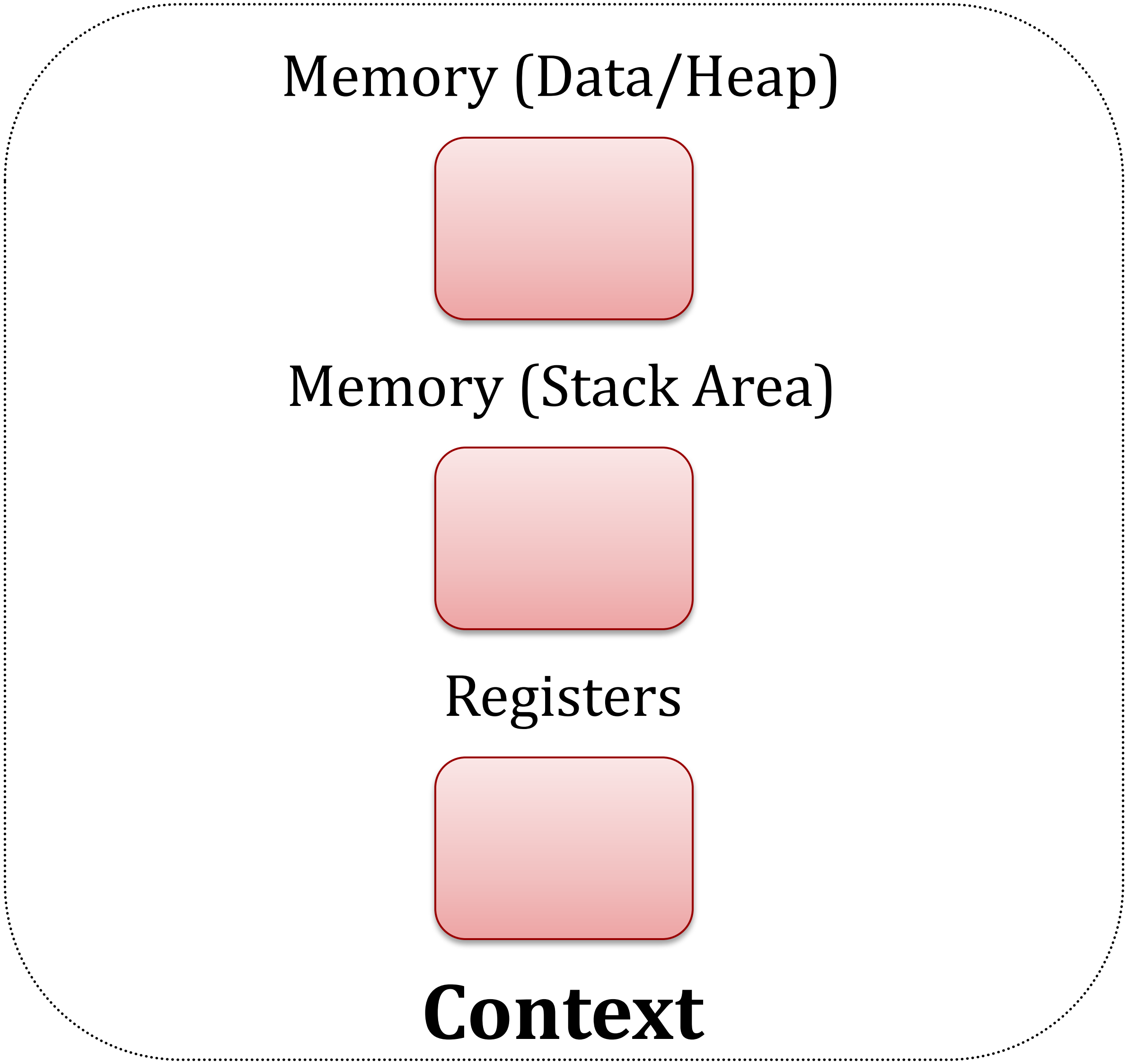
Each thread has its own code, program counter, set of registers, and stack

Threads



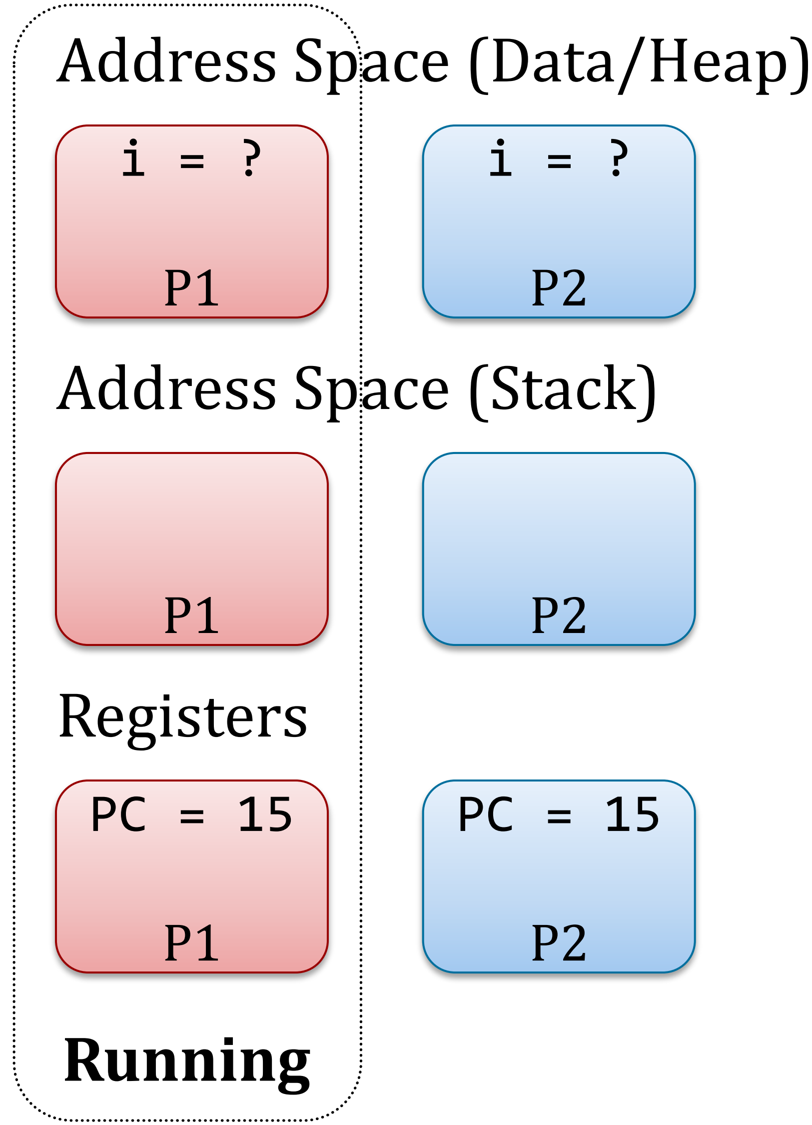
Process 1
Process 1
Process 1
User space
Kernel space
Thread
Kernel
(a)
Process
Thread
Kernel
(b)

Processes (Context)



Parallel Processes

```
1: int i;
2:
3: g()
4: {
5:     printf("Value of i is %d\n", i);
6: }
7:
8: f()
9: {
10:     g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:     return 0;
18: }
```



Parallel Processes

Input: 42

```
1: int i;
2:
3: g()
4: {
5:    printf("Value of i is %d\n", i);
6: }
7:
8: f()
9: {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```

Address Space (Data/Heap)

P1	P2
i = ***42***	i = ?

Address Space (Stack)

P1	P2

Registers

P1	P2
PC = ***16***	PC = 15

Running

Parallel Processes

```
1: int i;
2:
3: g()
4: {
5:     printf("Value of i is %d\n", i);
6: }
7:
8: f()
9: {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```

Address Space (Data/Heap)

P1	P2
i = 42	i = ?

Address Space (Stack)

P1	P2
17	

Registers

P1	P2
PC = *10*	PC = 15

Running

Parallel Processes

Input: 23

```
1:  int i;
2:
3:  g()
4:  {
5:     printf("Value of i is %d\n", i);
6:  }
7:
8:  f()
9:  {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```

Address Space (Data/Heap)

P1	P2
i = 42	i = ***23***

Address Space (Stack)

P1	P2
17	

Registers

P1	P2
PC = 10	PC = ***16***

Running

Parallel Processes

```
1: int i;
2:
3: g()
4: {
5:    printf("Value of i is %d\n", i);
6: }
7:
8: f()
9: {
10:   g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:   i = get_input();
16:   f();
17:   return 0;
18: }
```

Address Space (Data/Heap)

P1	P2
i = 42	i = 23

Address Space (Stack)

P1	P2
17	***17***

Registers

P1	P2
PC = 10	PC = ***10***

Running

Parallel Processes

```
1: int i;
2:
3: g()
4: {
5:     printf("Value of i is %d\n", i);
6: }
7:
8: f()
9: {
10:     g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:     return 0;
18: }
```

Address Space (Data/Heap)

P1	P2
i = 42	i = 23

Address Space (Stack)

P1	P2
17	17 ***11***

Registers

P1	P2
PC = 10	PC = ***5***

Running

Parallel Processes

```
1: int i;
2:
3: g()
4: {
5:     printf("Value of i is %d\n", i);
6: }
7:
8: f()
9: {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```

```
Value of i is 23
```

Address Space (Data/Heap)

P1	P2
i = 42	i = 23

Address Space (Stack)

P1	P2
17	17 11

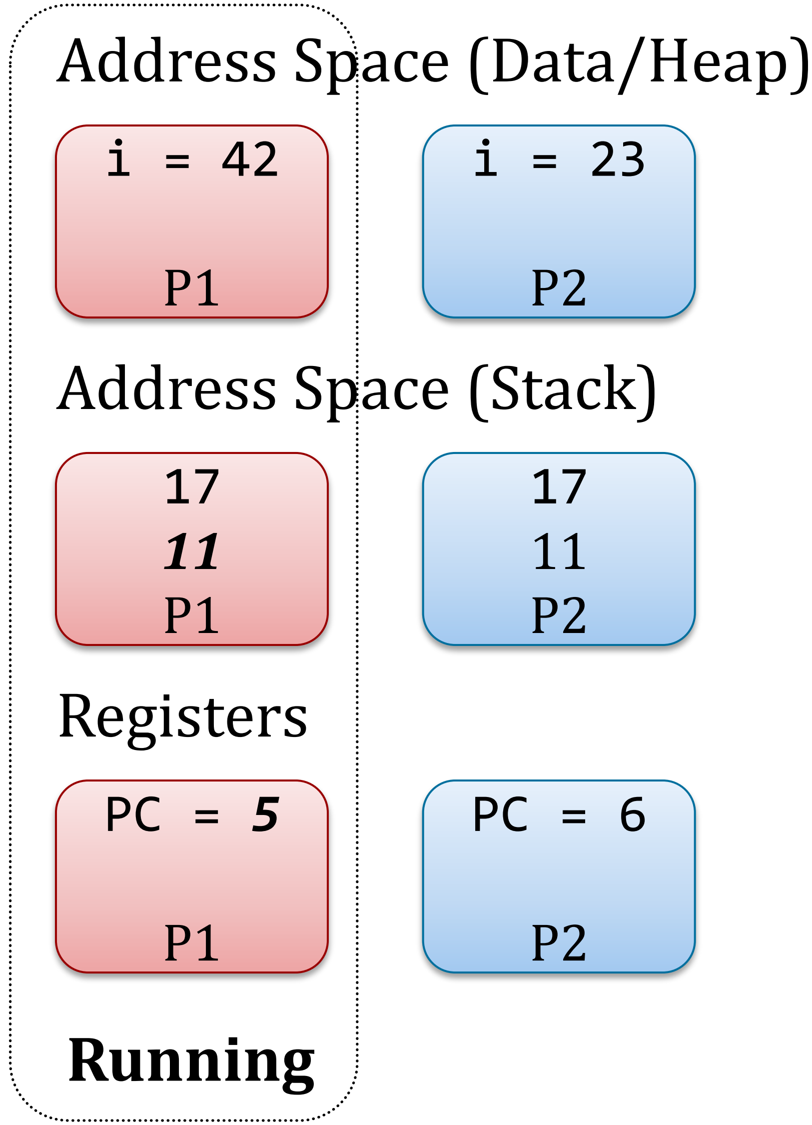
Registers

P1	P2
PC = 10	PC = ***6***

Running

Parallel Processes

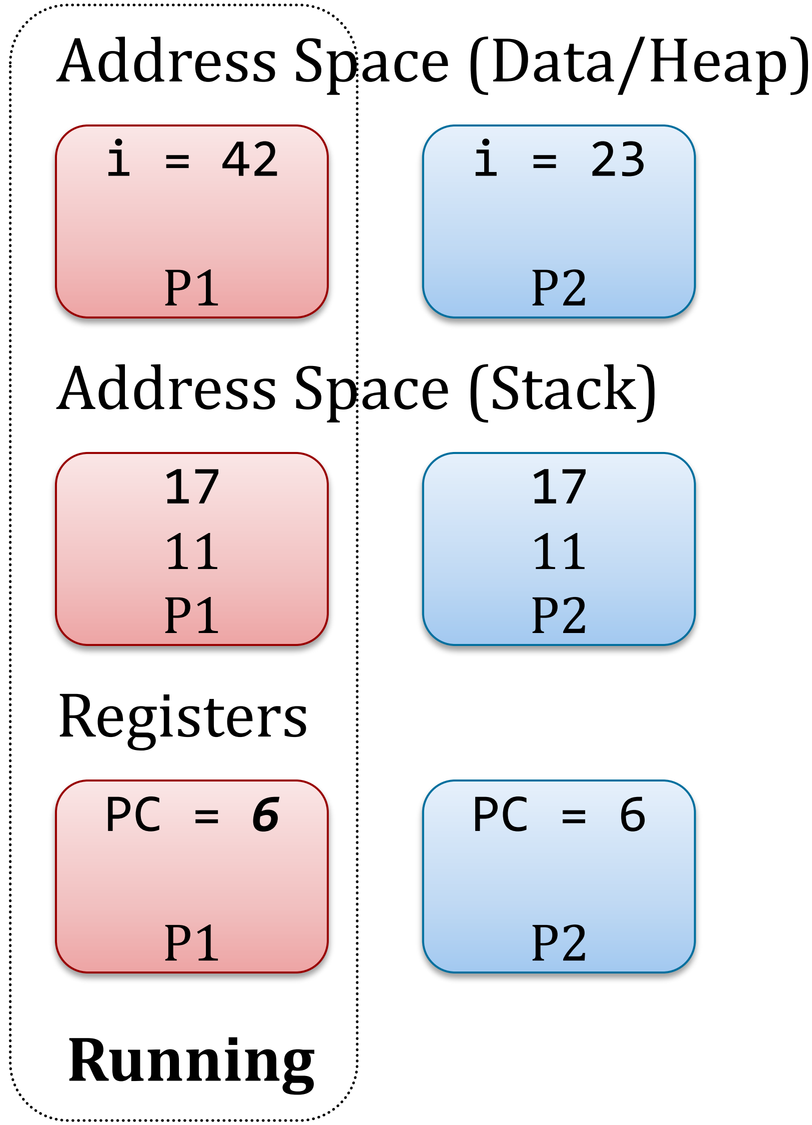
```
 1: int i;
 2:
 3: g()
 4: {
 5:     printf("Value of i is %d\n", i);
 6: }
 7:
 8: f()
 9: {
10:     g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:     return 0;
18: }
```



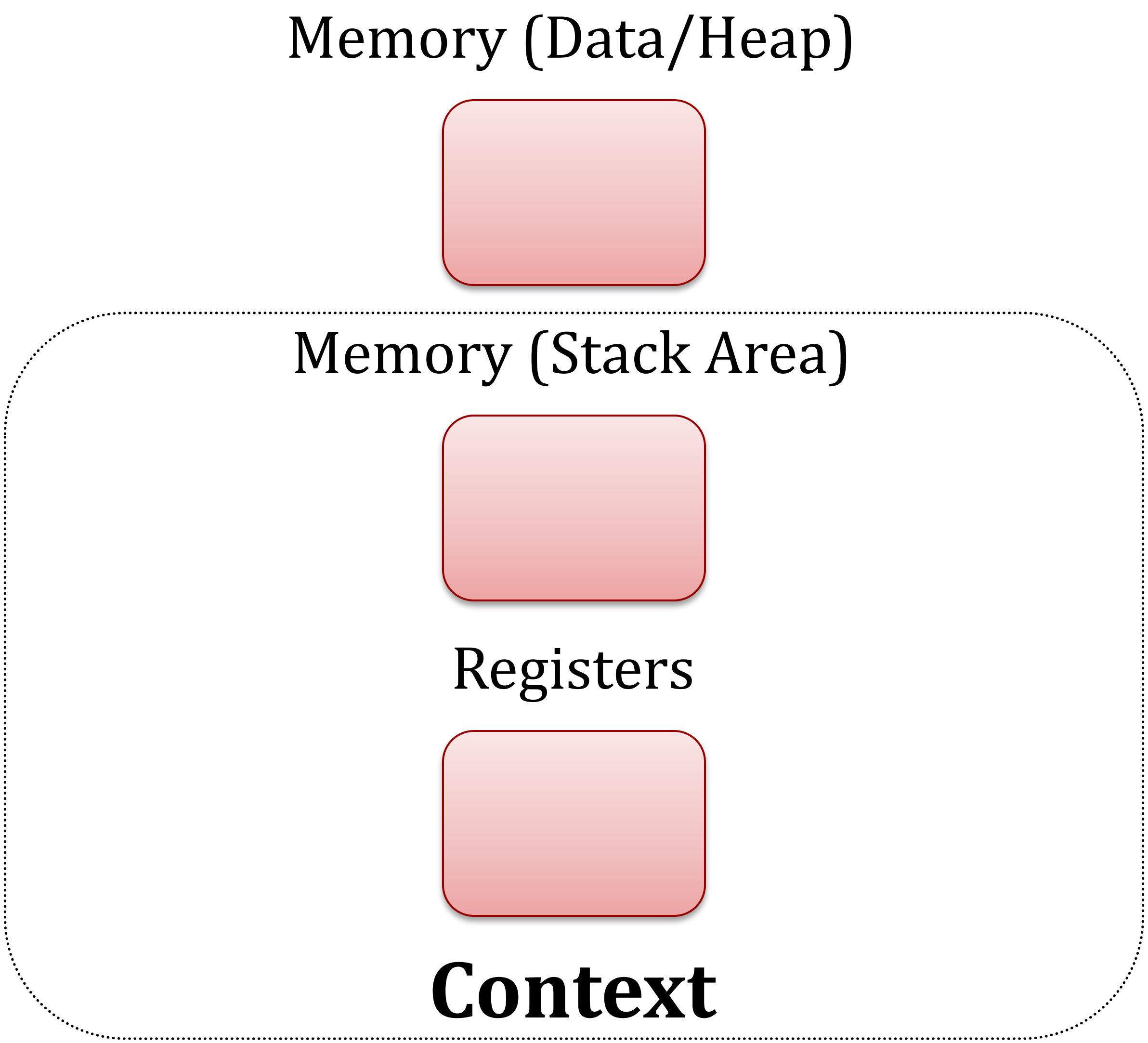
Parallel Processes

```
 1: int i;
 2:
 3: g()
 4: {
 5:    printf("Value of i is %d\n", i);
 6: }
 7:
 8: f()
 9: {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```

Value of i is 42

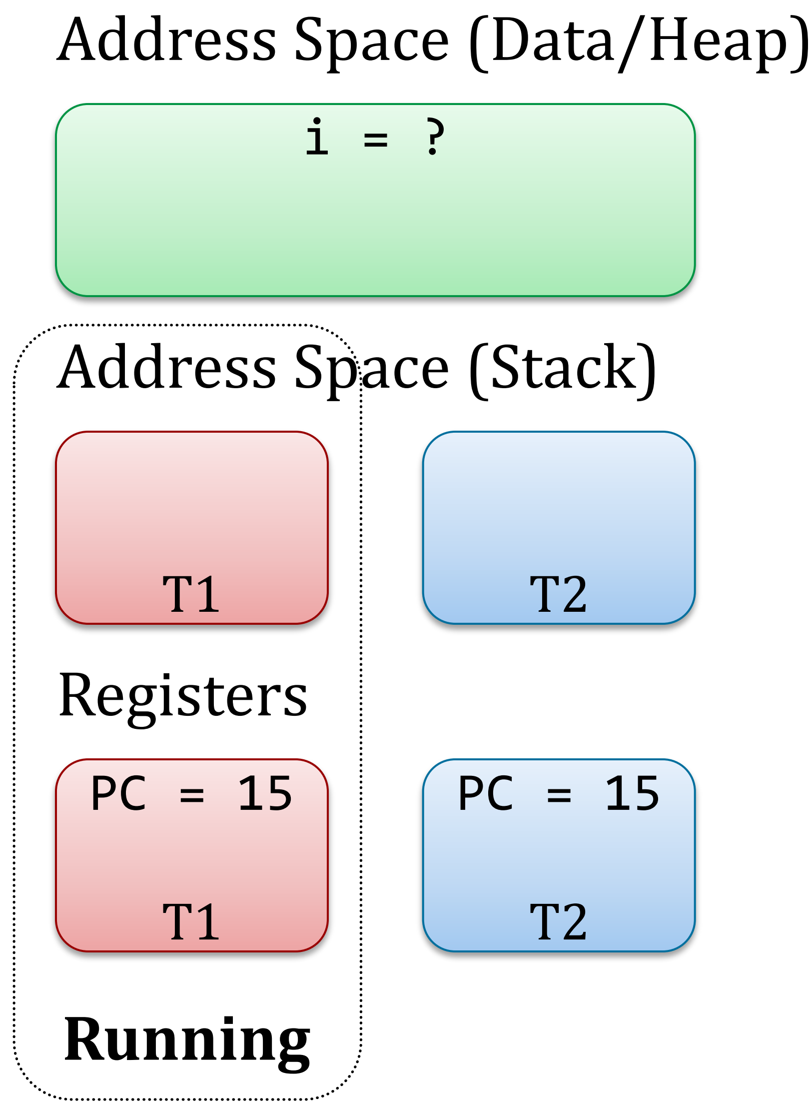


Threads (Context)



Threads

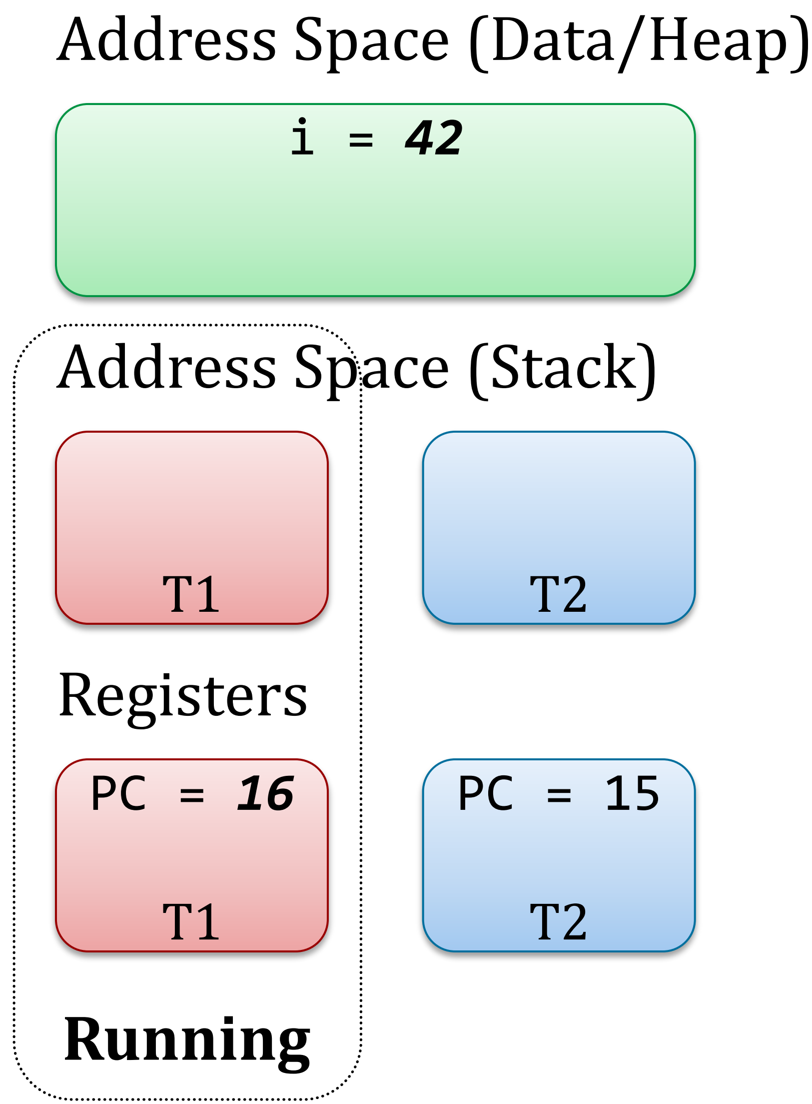
```
1:  int i;
2:
3:  g()
4:  {
5:     printf("Value of i is %d\n", i);
6:  }
7:
8:  f()
9:  {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```



Threads

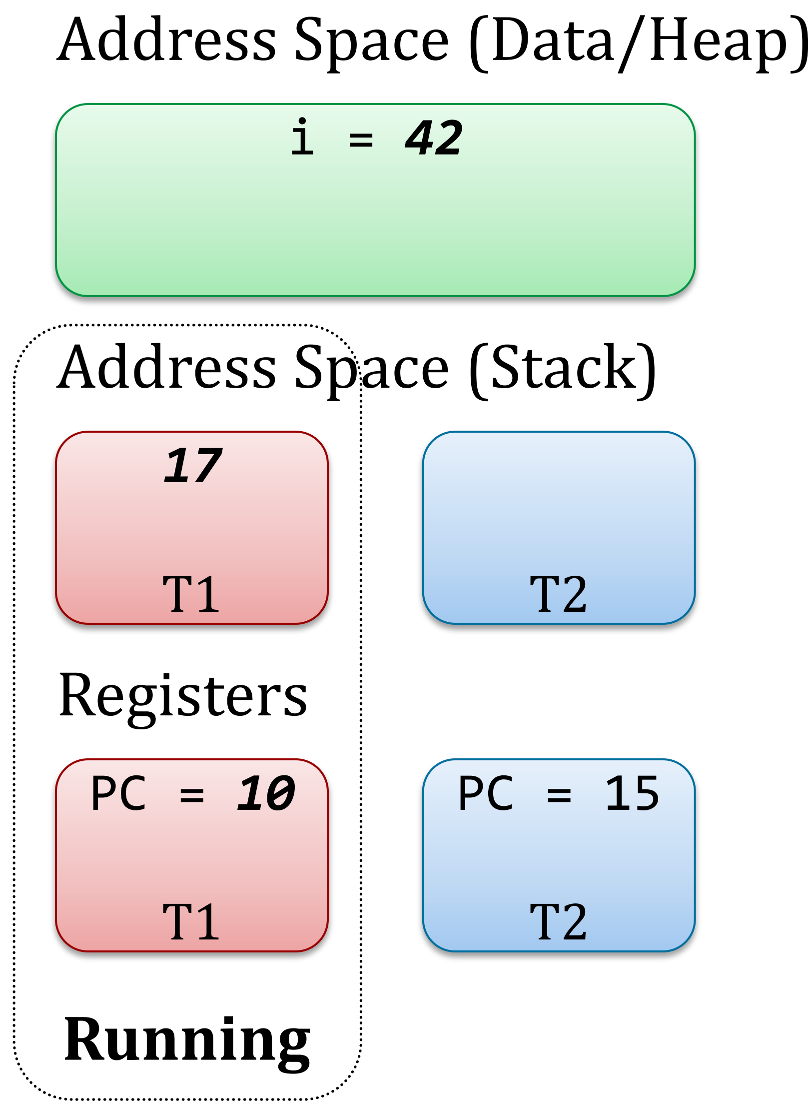
Input: 42

```
1:  int i;
2:
3:  g()
4:  {
5:     printf(“Value of i is %d\n“, i);
6:  }
7:
8:  f()
9:  {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```



Threads

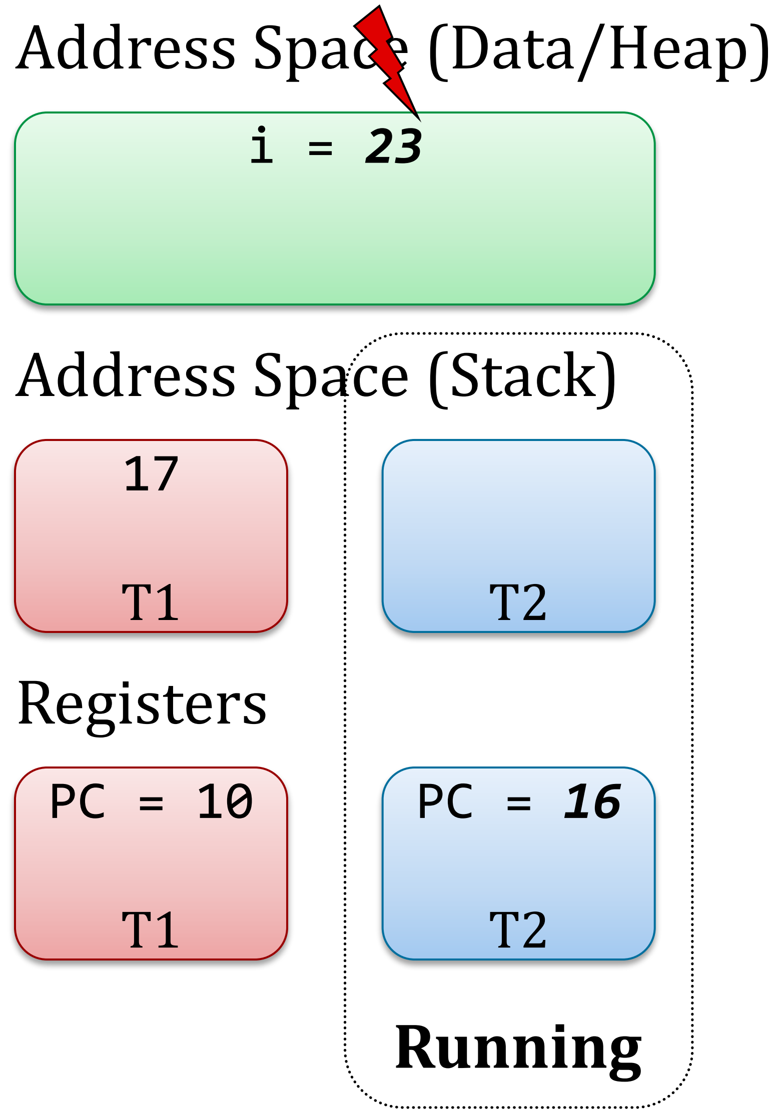
```
1: int i;
2:
3: g()
4: {
5:     printf("Value of i is %d\n", i);
6: }
7:
8: f()
9: {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```



Threads

Input: 23

```
 1: int i;
 2:
 3: g()
 4: {
 5:    printf("Value of i is %d\n", i);
 6: }
 7:
 8: f()
 9: {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```



Threads

```
1: int i;
2:
3: g()
4: {
5:     printf("Value of i is %d\n", i);
6: }
7:
8: f()
9: {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```

Address Space (Data/Heap)

i = 23

Address Space (Stack)

T1	T2
17	***17***

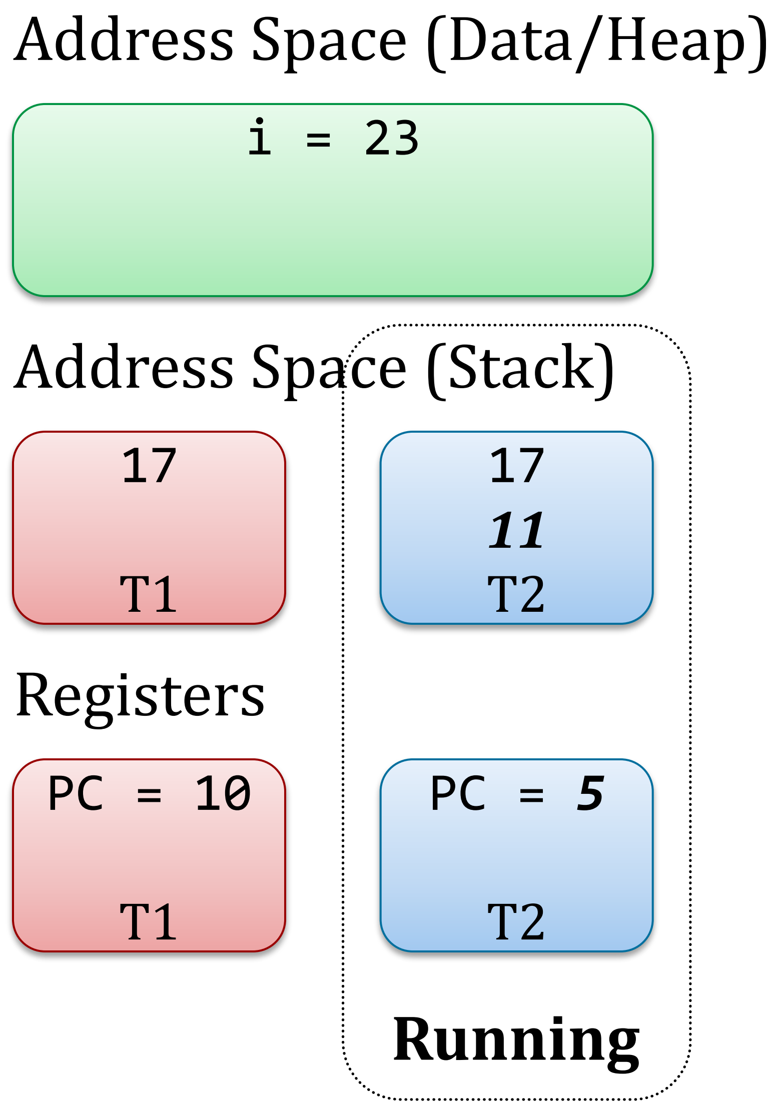
Registers

T1	T2
PC = 10	PC = ***10***

Running

Threads

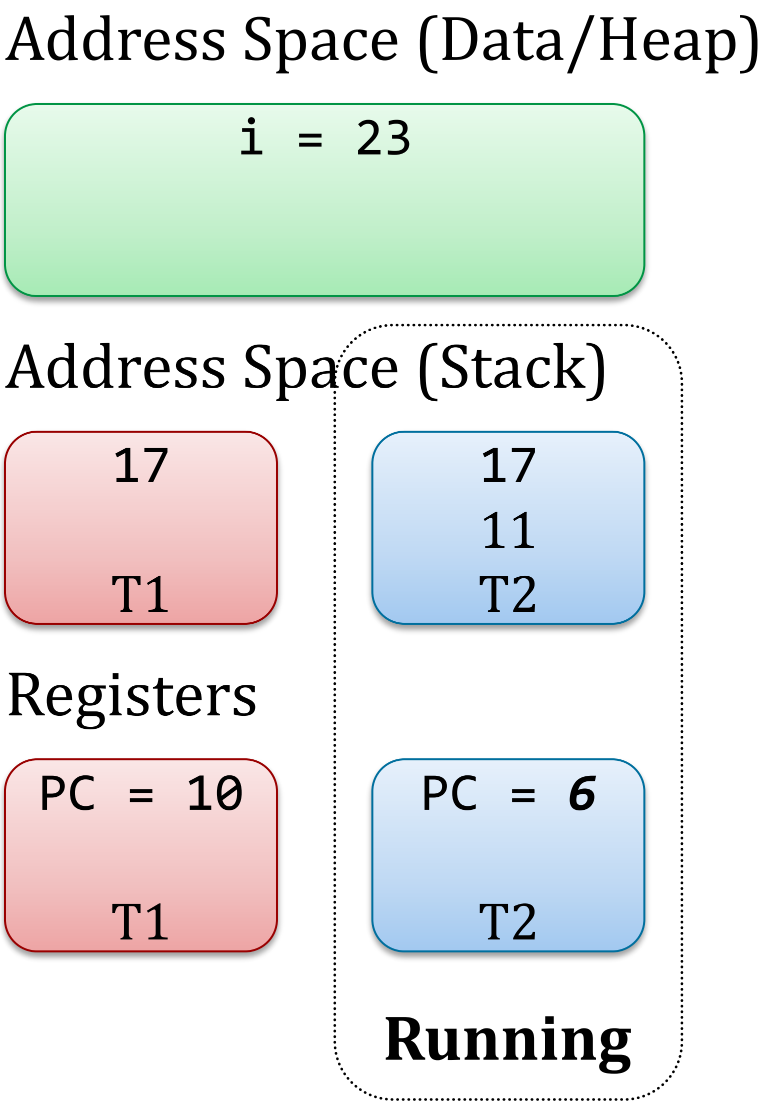
```
1: int i;
2:
3: g()
4: {
5:     printf("Value of i is %d\n", i);
6: }
7:
8: f()
9: {
10:     g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:     return 0;
18: }
```



Threads

```
1: int i;
2:
3: g()
4: {
5:    printf("Value of i is %d\n", i);
6: }
7:
8: f()
9: {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```

Value of i is 23



Threads

```
1: int i;
2:
3: g()
4: {
5:     printf("Value of i is %d\n", i);
6: }
7:
8: f()
9: {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```

Address Space (Data/Heap)

i = 23

Address Space (Stack)

T1	T2
17	17
11	11

Registers

T1	T2
PC = ***5***	PC = 6

Running

Threads

```
1:  int i;
2:
3:  g()
4:  {
5:     printf("Value of i is %d\n", i);
6:  }
7:
8:  f()
9:  {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```

Value of i is 23

Address Space (Data/Heap)

i = 23

Address Space (Stack)

T1	T2
17	17
11	11
T1	T2

Registers

T1	T2
PC = ***6***	PC = 6
T1	T2

Running

Why Threads?

Useful to structure applications that have to do many things concurrently

- One thread is waiting for I/O
- Another thread *in the same process* is doing some computation
- Examples: Web Browser (rendering & network), Web Server (accepting connections & answering requests)

Having threads share common address space makes it easier to coordinate activities

Use a shared data-structure through which the processes can be coordinated

- Producer-Consumer interactions
- Shared data structures/counters

More efficient than using processes

- Why?
- Context switch is faster!

Thread Primitives

- <u>**`thread_create()`**</u>
- <u>**`thread_exit()`**</u>
- <u>**`thread_join()`**</u>
- <u>**`thread_yield()`**</u>

(+ synchronization primitives)

Thread Implementation

Threads can be implemented in user space

Pros

- Performance (no kernel/user switch)
- Portability (same primitives for every environment)
- Flexibility (custom scheduling algorithm)

Cons

- Blocking system calls block the process, not the thread
  - need to check if a system call would block before each invocation
- Threads cannot be easily preempted (they have to *yield*)

Thread Implementation

Threads can be implemented in the kernel

Pros

- Blocking system calls suspend the calling thread only
- Signals can be delivered more precisely

Cons

- Can be heavy, not as flexible

Threading Issues

- **What happens on a fork()?**
  - only a single thread is created in the child
- **What happens with shared data structures and files?**
  - threads need to be careful and synchronize access
- **What about stack management?**
  - each thread needs its own stack
- **What about signal delivery?**
  - complicated!
  - some signals are sent to specific thread (alarm, segfault)
  - others to the first that does not block them (termination request)

Reentrant Functions

What about global variables in libraries?

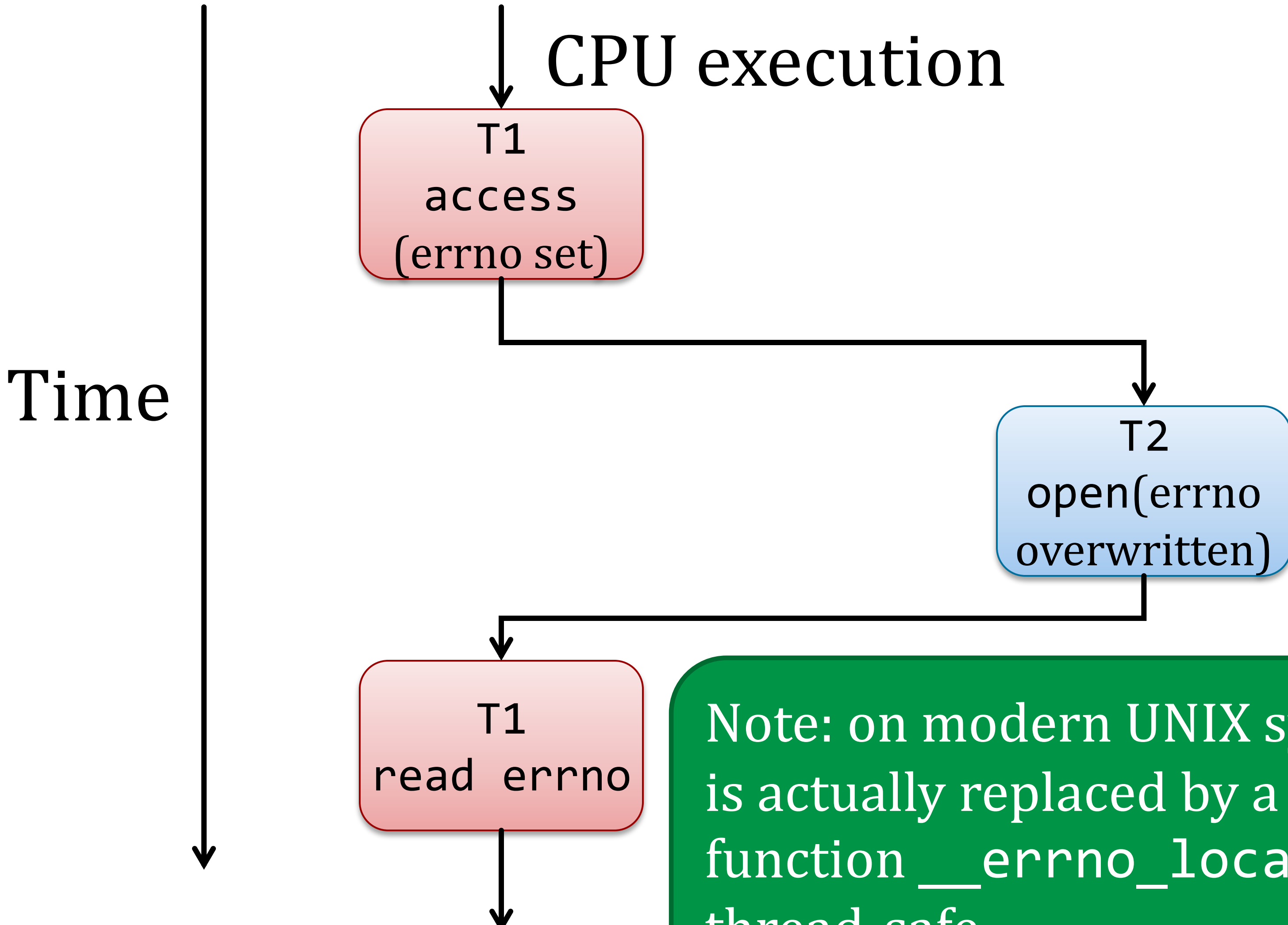
– functions need to be reentrant

Some functions are not designed to be invoked concurrently

– Use of global variables, such as *errno*

Functions used by threads need to be *reentrant*

errno Example



Note: on modern UNIX systems, `errno` is actually replaced by a call to a function `__errno_location()` that is thread-safe

Portability Issues and Pthreads

- POSIX 1003.1c (a.k.a. pthreads) is an API for multi-threaded programming standardized by IEEE as part of the POSIX standards
- Most Unix vendors have endorsed the POSIX 1003.1c standard
- Implementations of 1003.1c API are available for many UNIX systems
- pthreads defines an interface
  - implementation can be done in either user or kernel space
- Thus, multithreaded programs using the 1003.1c API are likely to run unchanged on a wide variety of Unix platforms