EC 440 – Introduction to Operating Systems

Orran Krieger (BU)
Larry Woodman (Red Hat)

Administration and Review

Review

- Process and related system calls
- Signals and related system calls
- Memory layout and related system calls
- Files and related system calls
- pipes, dup and how to exec two programs that can talk to each other

We are now proposing reading, see piazza: Chapter 2 for the next two lectures

Project 1 – One More Week

If you are having difficulties

- go to office hours
- use Piazza

Some comments:

- use multiple functions, compiled into separate object files
- proper Makefile (dependencies)
- use your debugger, not printf
- check return arguments
- perror/exit
- compile with -Wall

You will save yourself infinite amounts of time… especially for the more complex project coming up

This lecture

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
- putting it together: 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;
}
```

Process Life Cycle

A process can be created

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

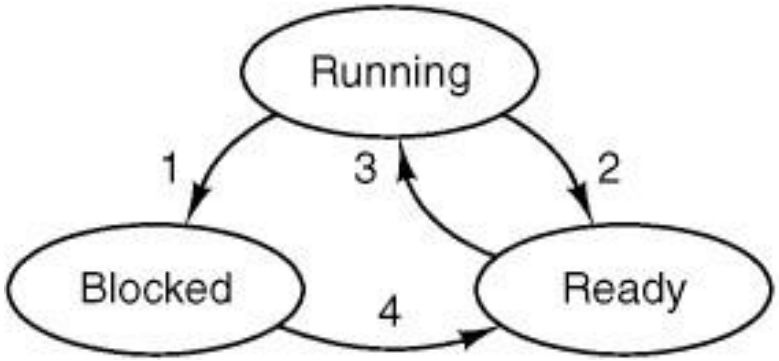
A process can be terminated

- 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.

Generic Process Implementation

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

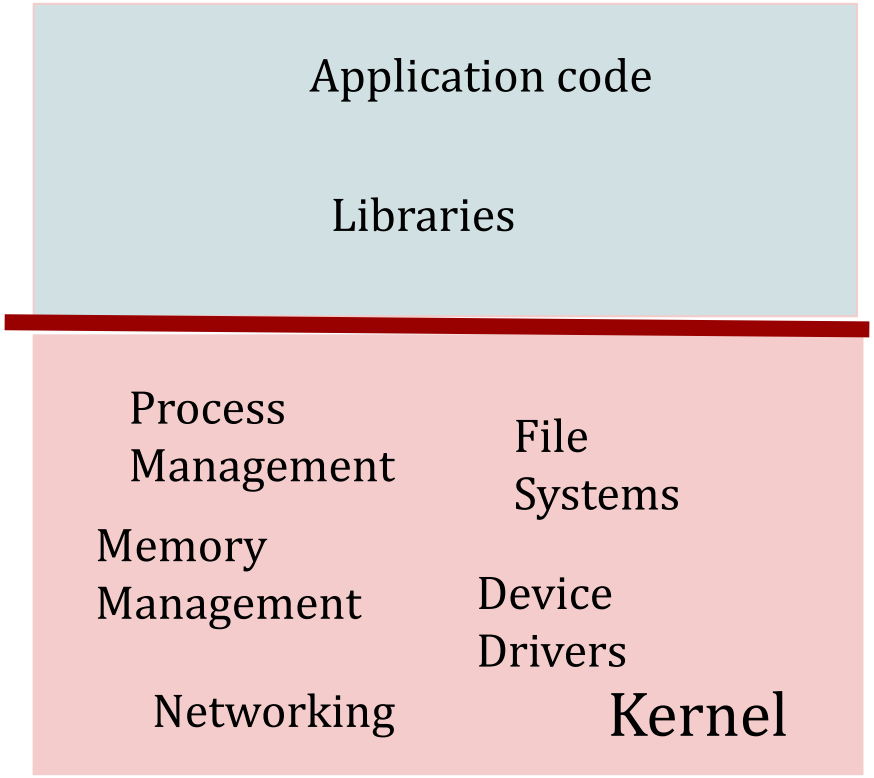
The PCB contains:

- 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

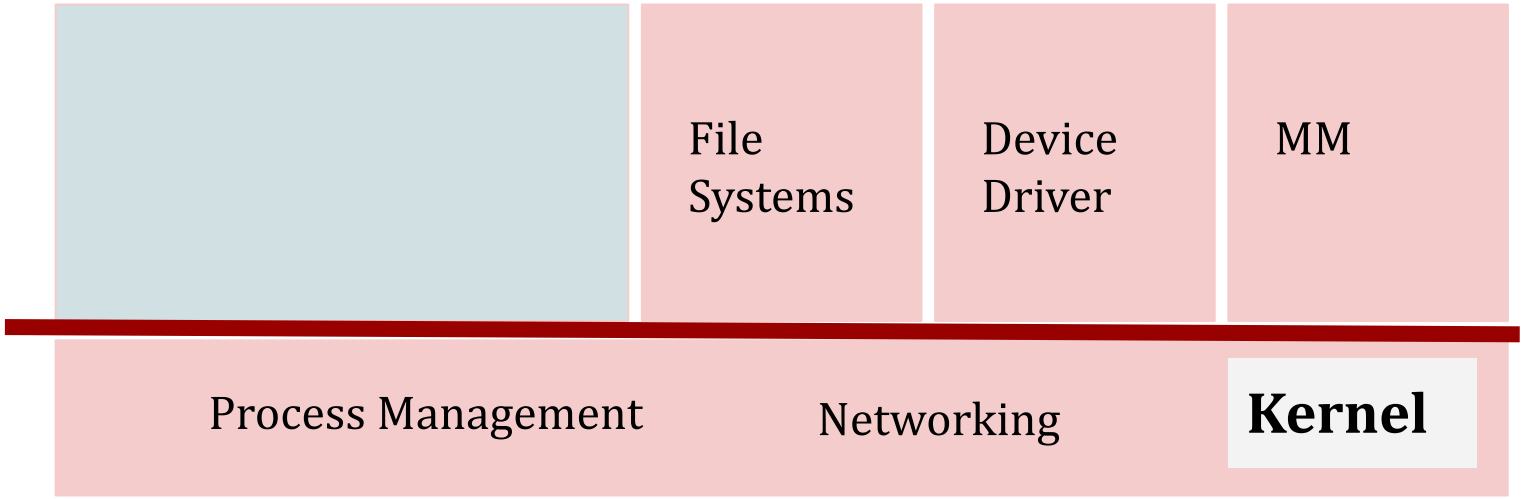
PCB In Linux – task_struct

```
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 ...
}
```

That's great for monolithic OSes



More challenging for other designs like micro-kernels



Consider micro-kernel's like Minux

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)

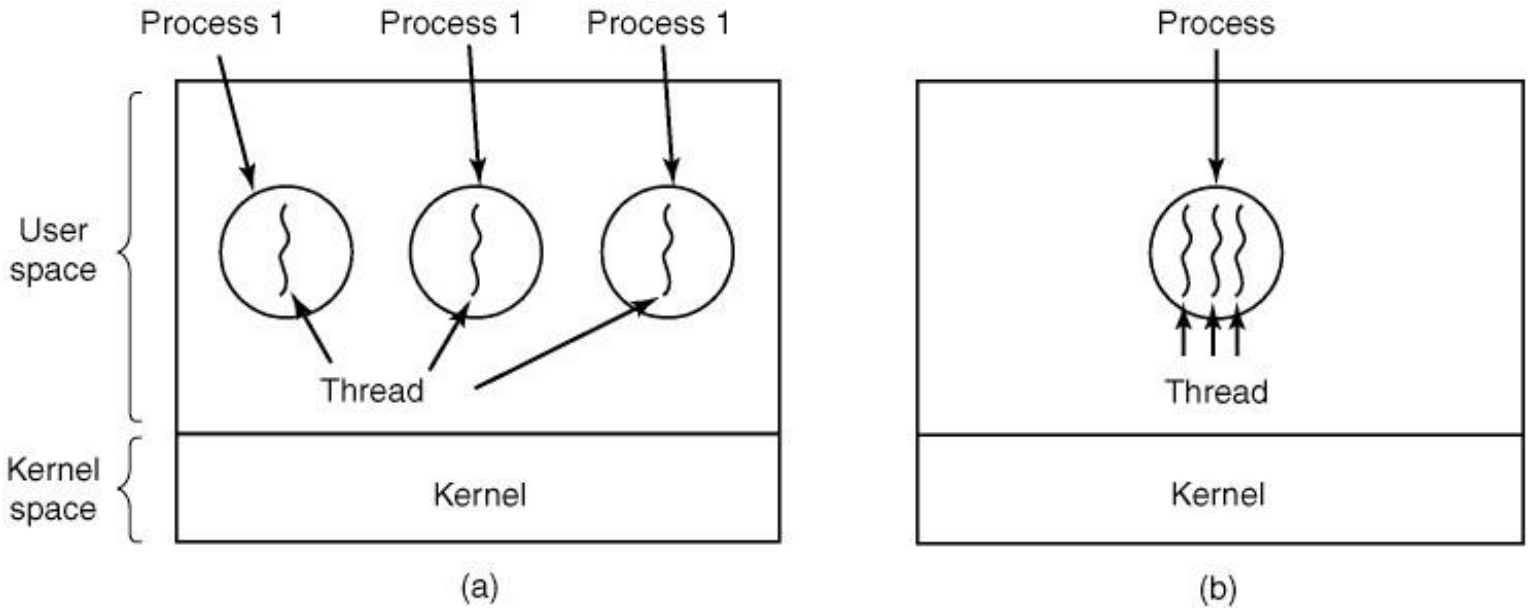
Multiple threads of execution can run in the same process environment

Multiple threads (in the same process) share

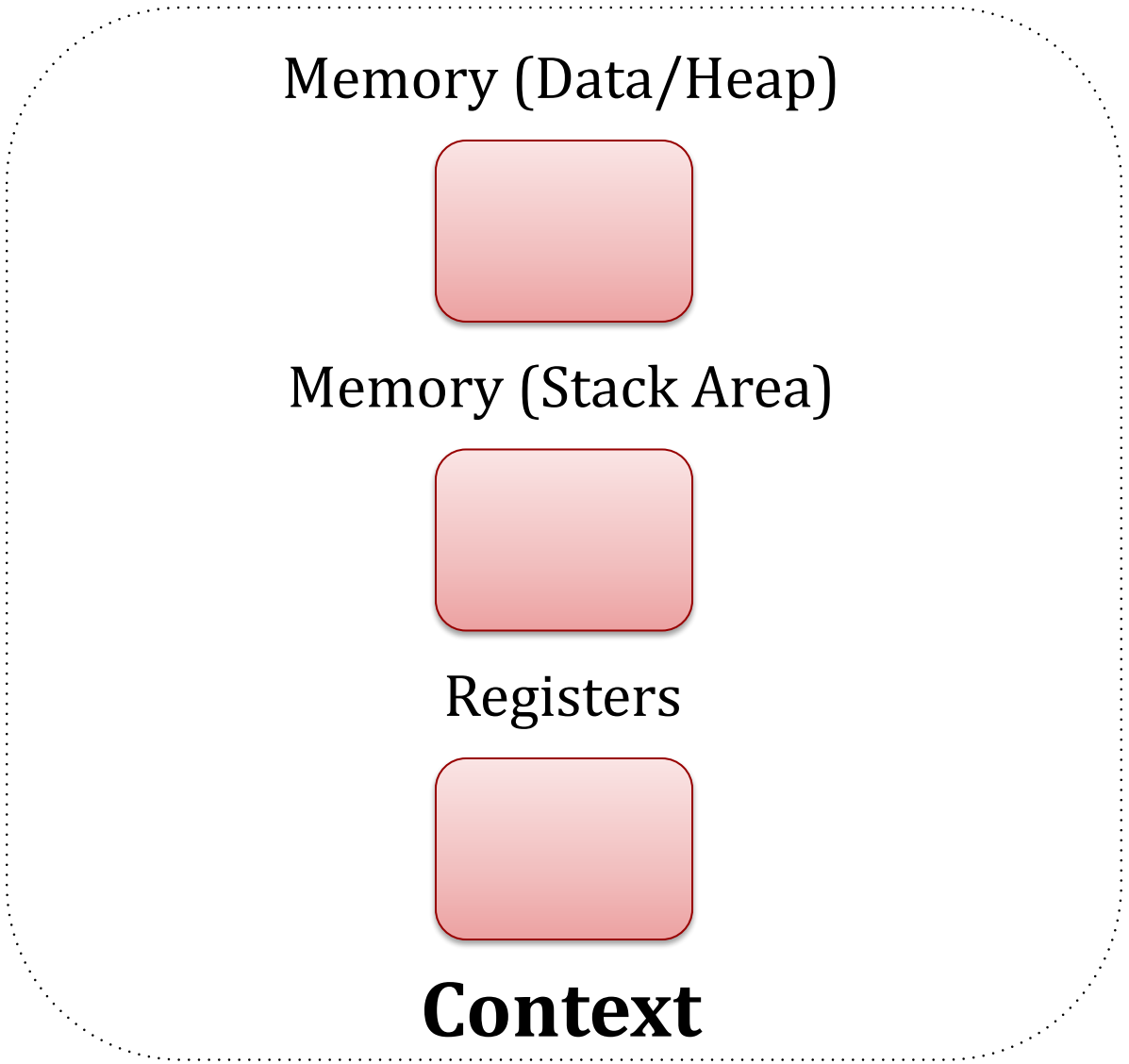
- Common address space (shared memory)
- Open files
- Process, user, and group IDs

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

Threads

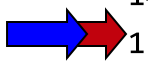

Process 1
Process 1
Process 1
User space
Thread
Kernel space
Kernel
(a)
Process
Thread
Kernel
(b)

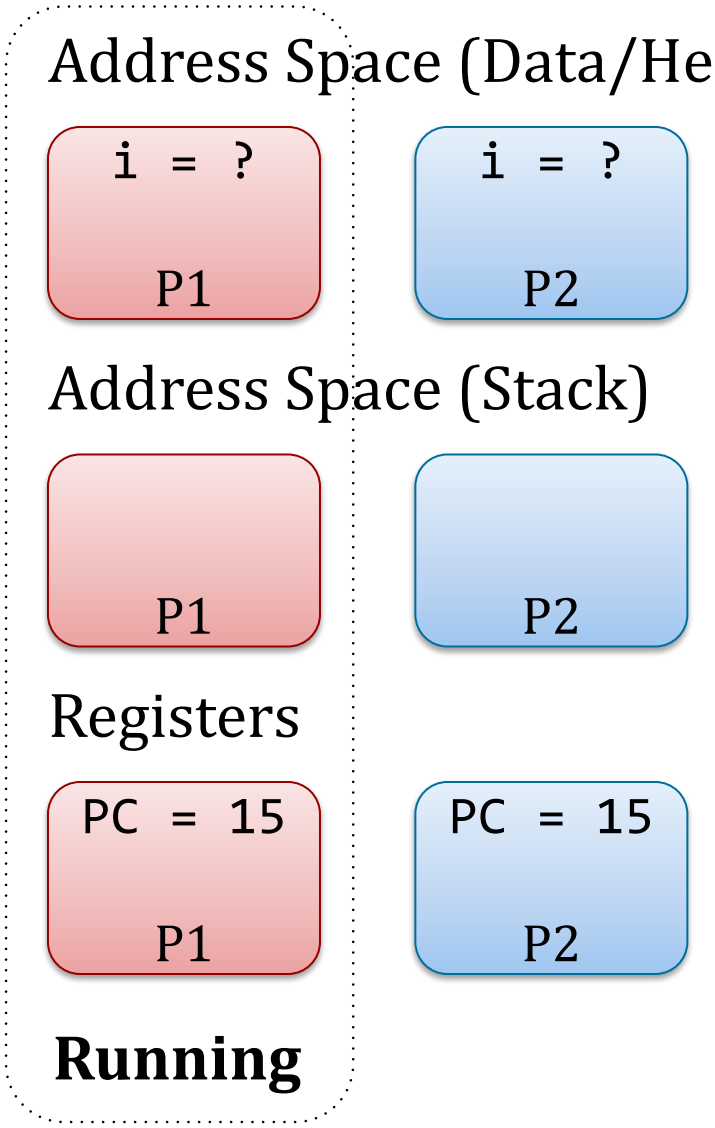
Processes (Context)



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: }
```

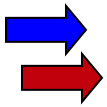


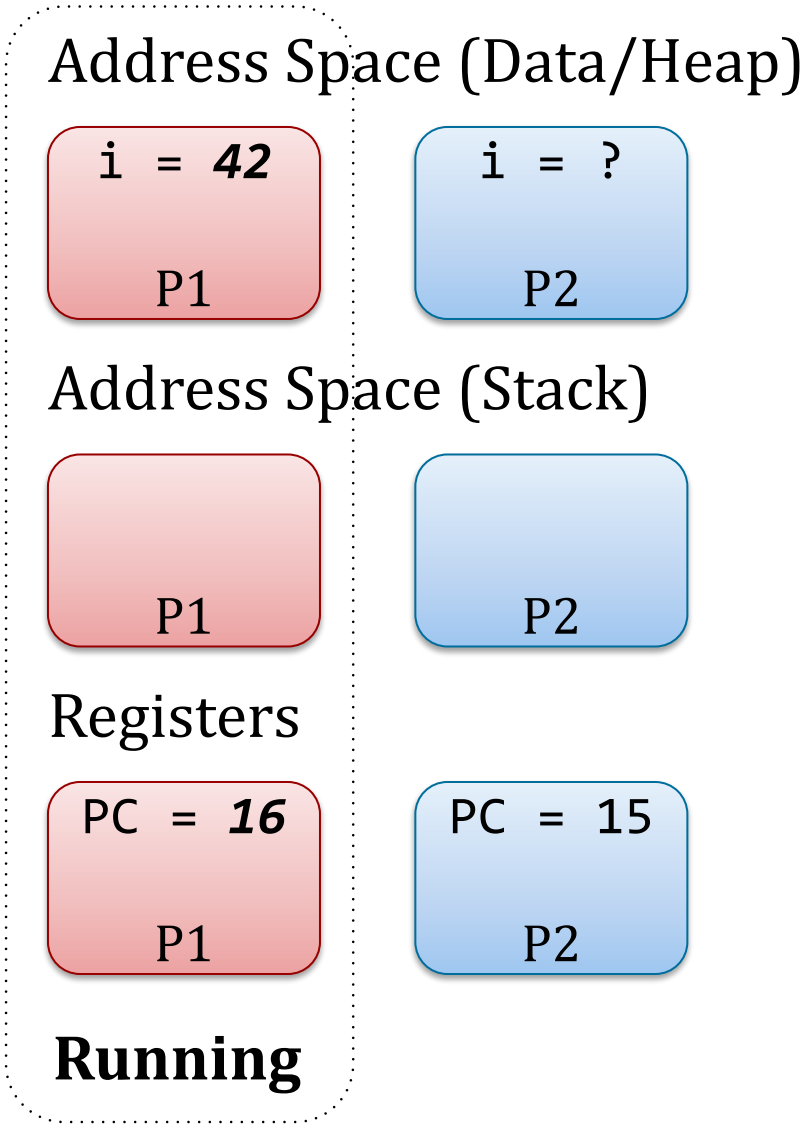


Parallel Processes

Input: 42

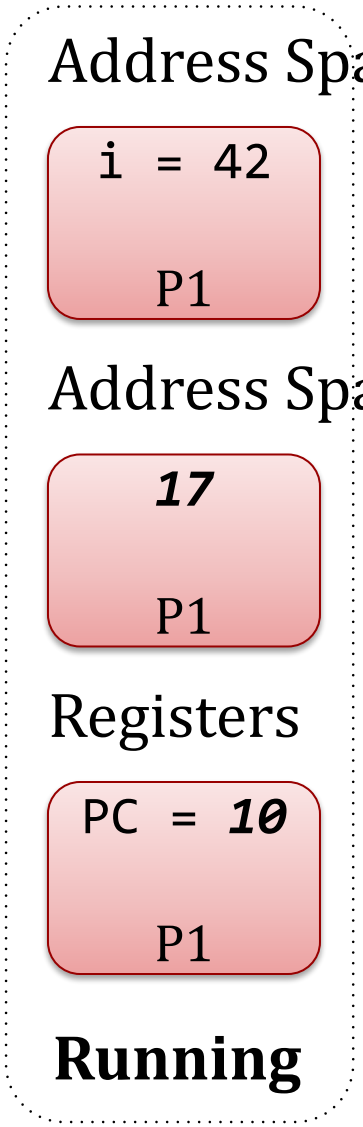
```
 1: int i;
 2:
 3: g()
 4: {
 5:     printf("Value of i is %d\n", i);
 6: }
 7:
 8: f()
 9: {
10:      g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:      return 0;
18: }
```





Parallel Processes

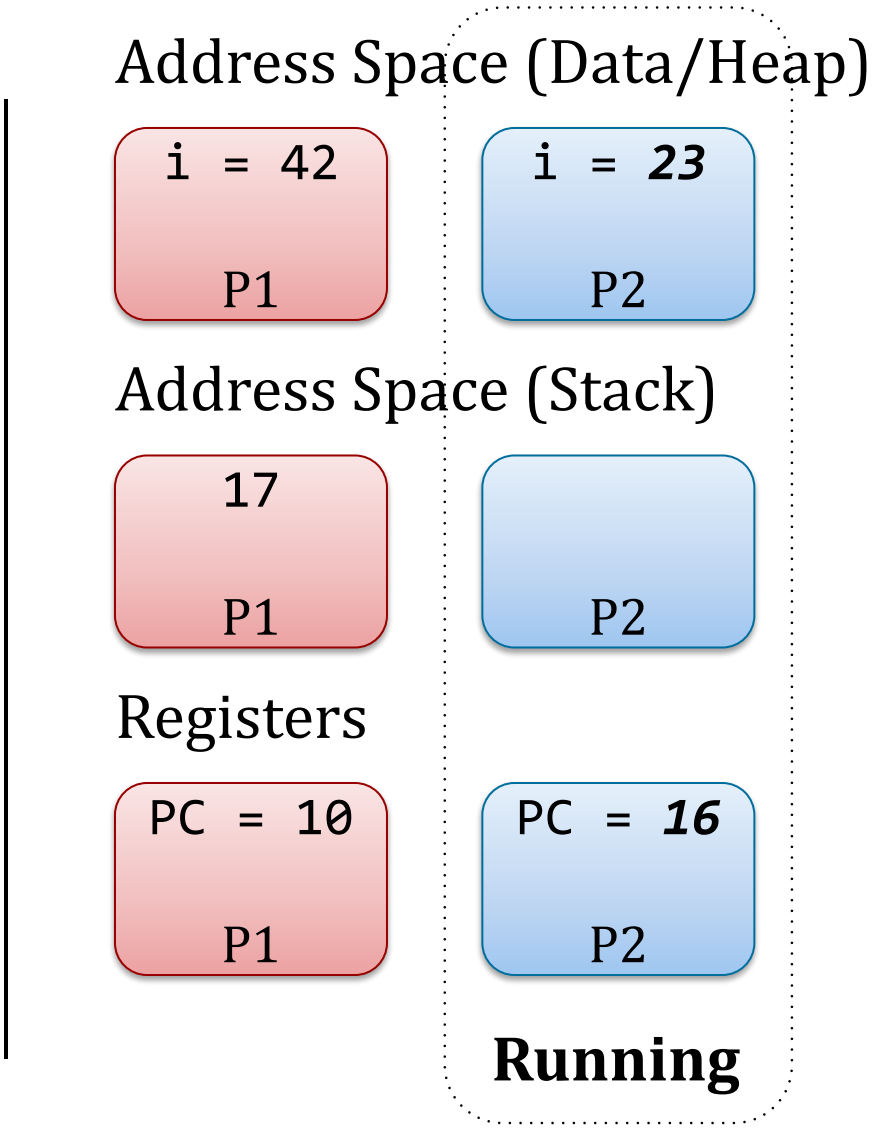
```
1: int i;
2:
3: g()
4: {
5:     printf("Value of i is %d\n", i);
6: }
7:
8: f()
9: {
10:      g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:       return 0;
18: }
```



Parallel Processes

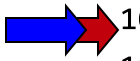
Input: 23

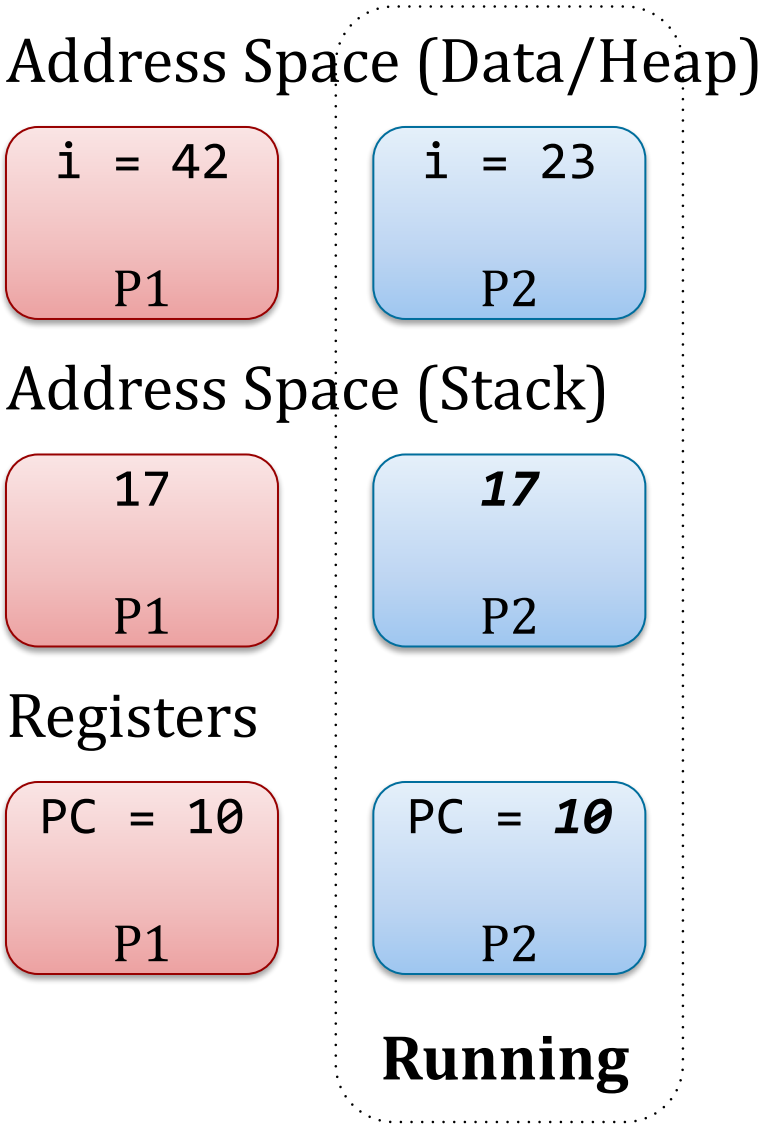
```
1: int i;
2:
3: g()
4: {
5:     printf("Value of i is %d\n", i);
6: }
7:
8: f()
9: {
10:      g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:      return 0;
18: }
```



Parallel Processes

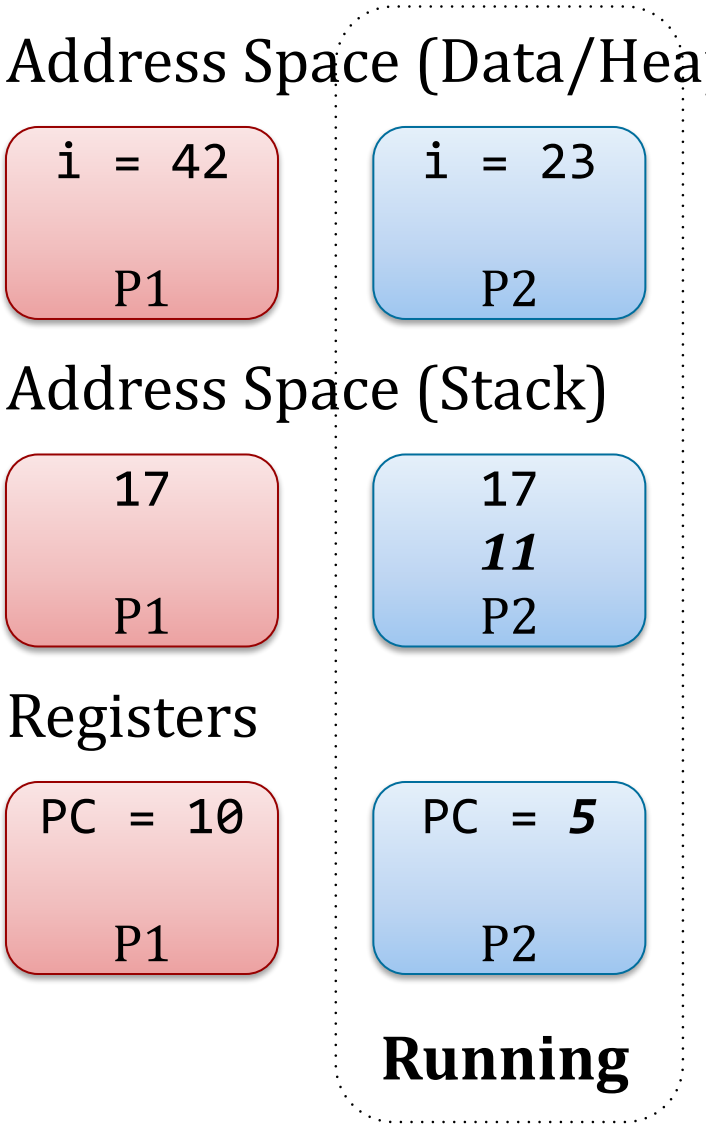
```
1: int i;
2:
3: g()
4: {
5:     printf("Value of i is %d\n", i);
6: }
7:
8: f()
9: {
10:      g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:      return 0;
18: }
```





Parallel Processes

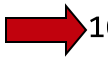
```
1: int i;
2:
3: g()
4: {
5:     printf("Value of i is %d\n", i);
6: }
7:
8: f()
9: {
10:      g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:       return 0;
18: }
```



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)

i = 42	i = 23
P1	P2

Address Space (Stack)

17	17
P1	11
	P2

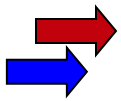
Registers

PC = 10	PC = *6*
P1	P2

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 ***11*** P1	17 11 P2

Registers

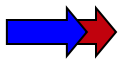
P1	P2
PC = ***5***	PC = 6

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: }
```

Value of i is 42



Address Space (Data/Heap)

i = 42	i = 23
P1	P2

Address Space (Stack)

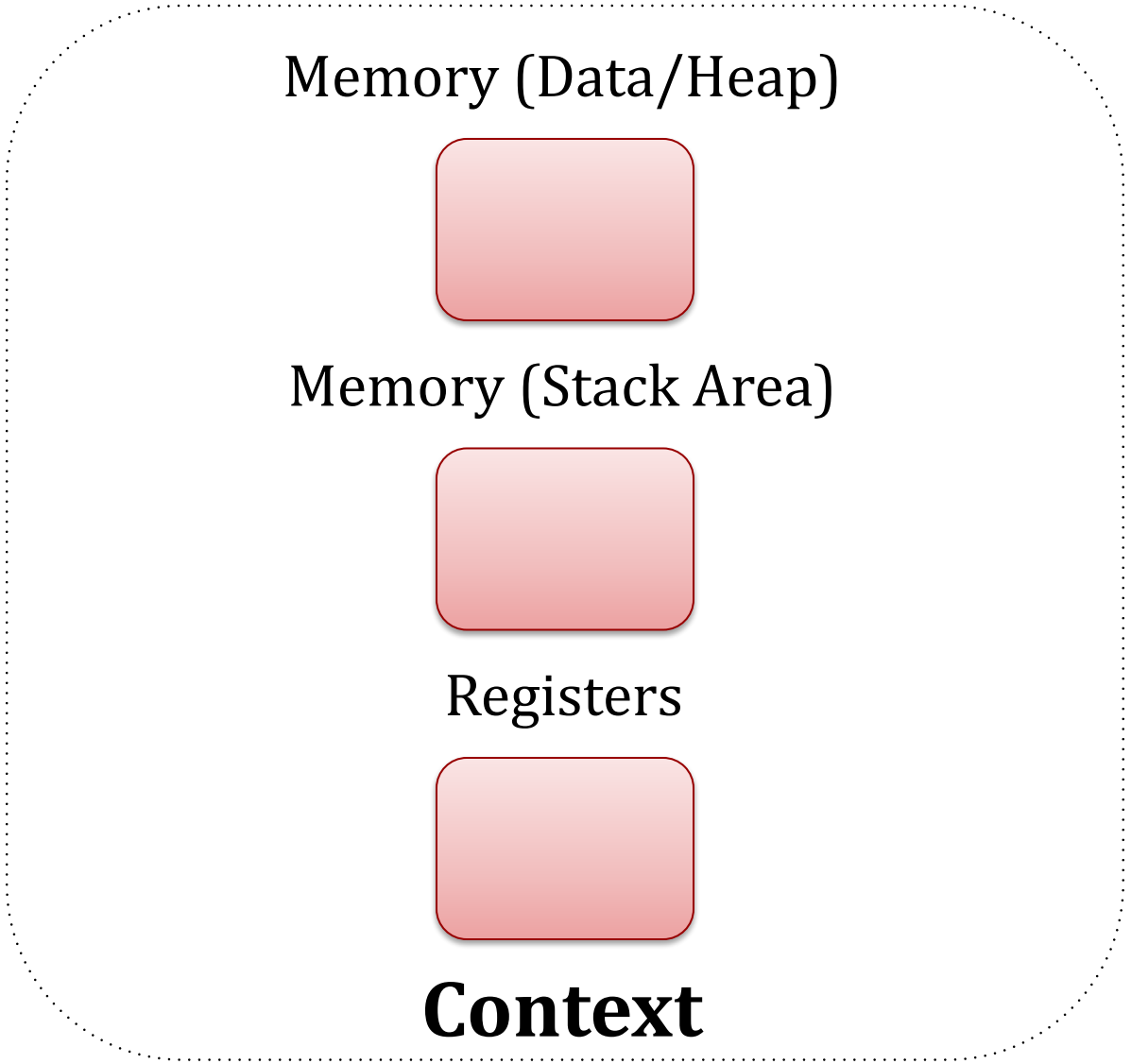
17 11 P1	17 11 P2

Registers

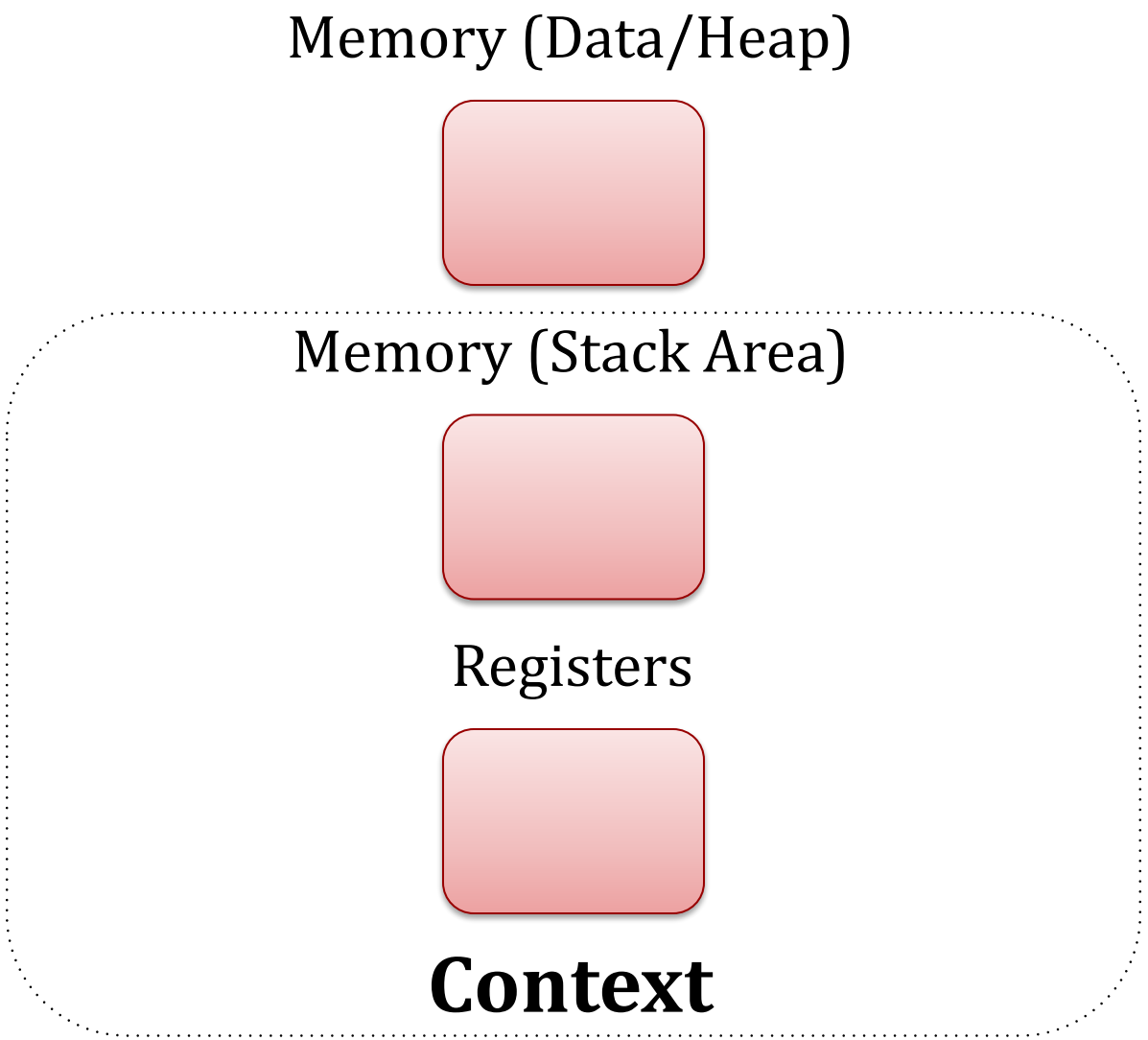
PC = ***6***	PC = 6
P1	P2

Running

Processes (Context)

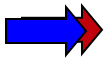


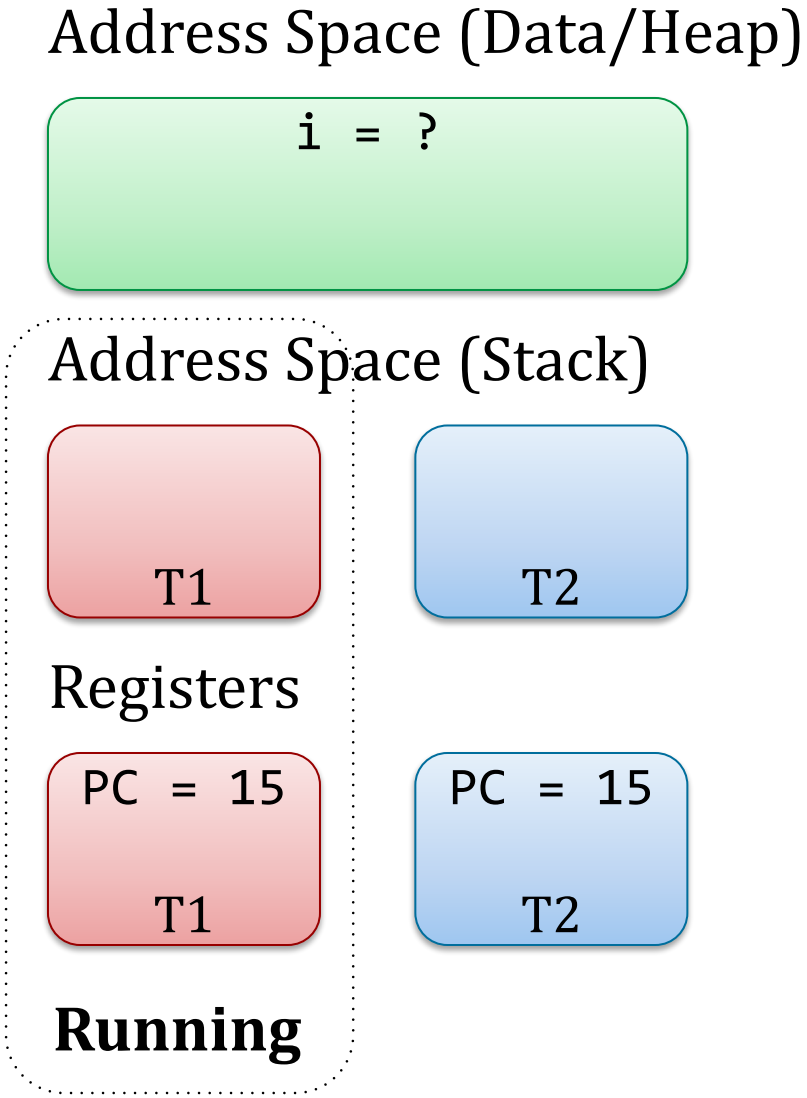
Threads (Context)



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: }
```

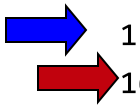




Threads

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: }
```



Address Space (Data/Heap)

i = *42*

Address Space (Stack)

T1

T2

Registers

PC = *16*

T1

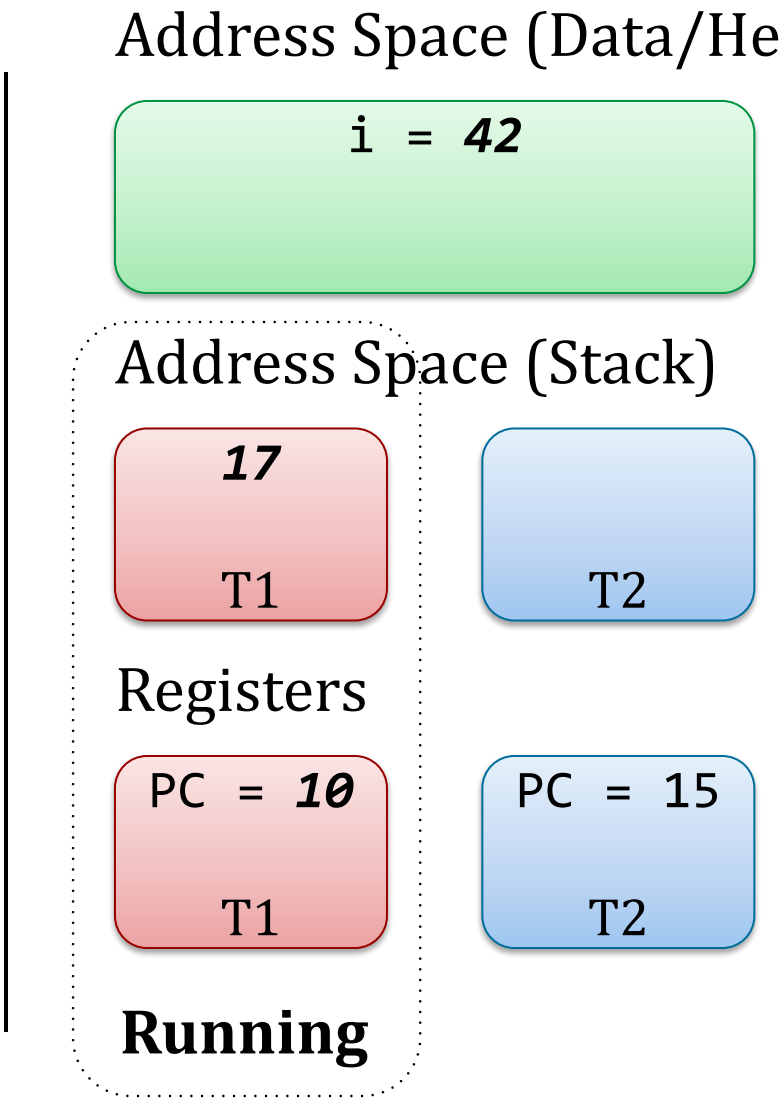
PC = 15

T2

Running

Threads

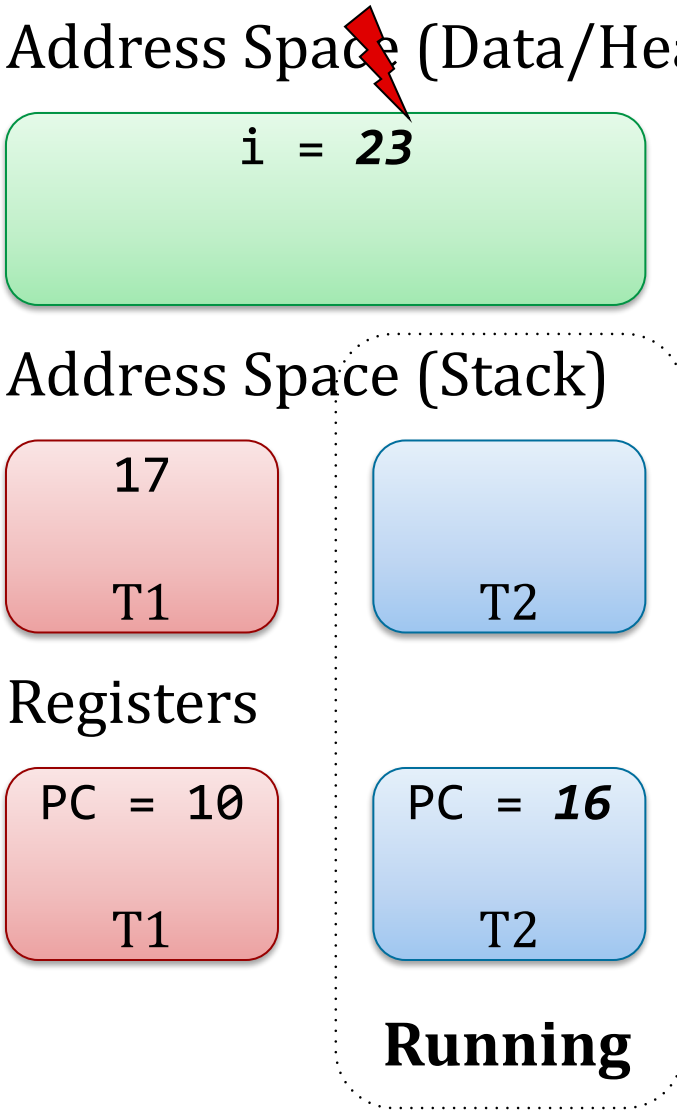
```
1: int i;
2:
3: g()
4: {
5:     printf("Value of i is %d\n", i);
6: }
7:
8: f()
9: {
10:      g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:      return 0;
18: }
```



Threads

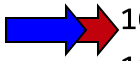
Input: 23

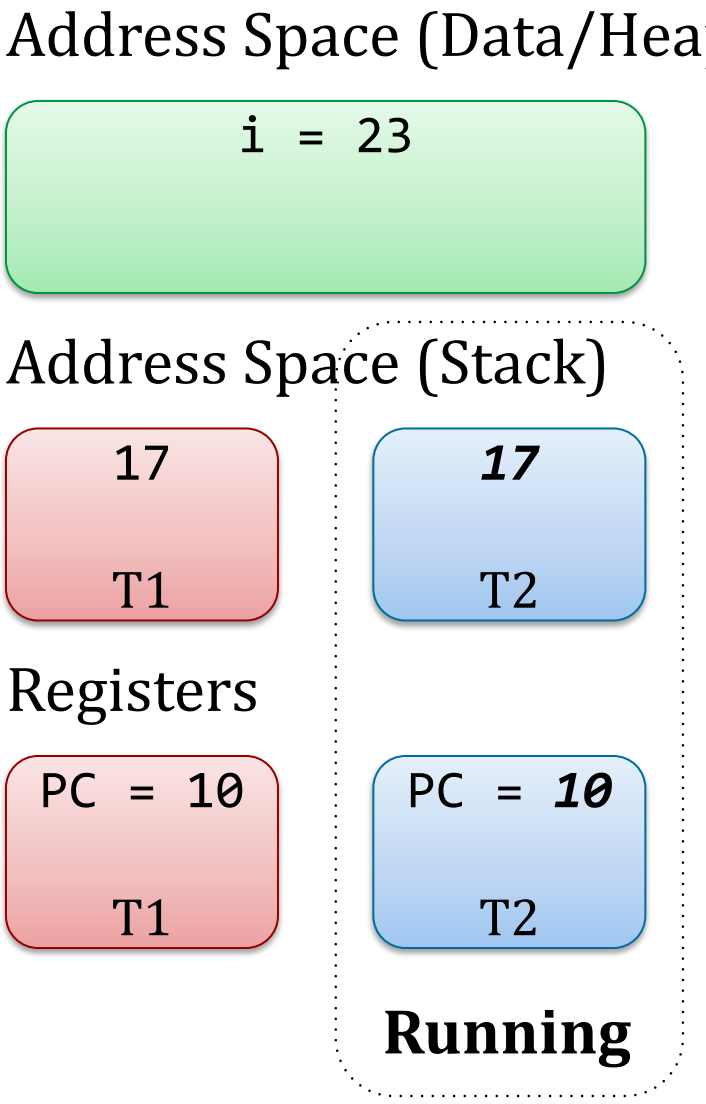
```
1: int i;
2:
3: g()
4: {
5:     printf("Value of i is %d\n", i);
6: }
7:
8: f()
9: {
10:     g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:     return 0;
18: }
```



Threads

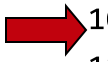
```
1: int i;
2:
3: g()
4: {
5:     printf("Value of i is %d\n", i);
6: }
7:
8: f()
9: {
10:      g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:      return 0;
18: }
```

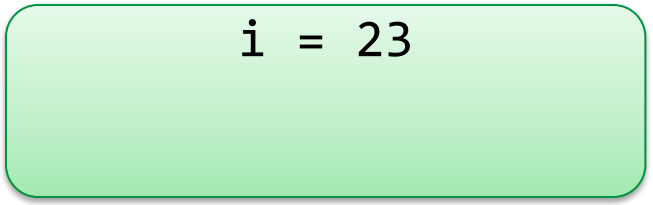


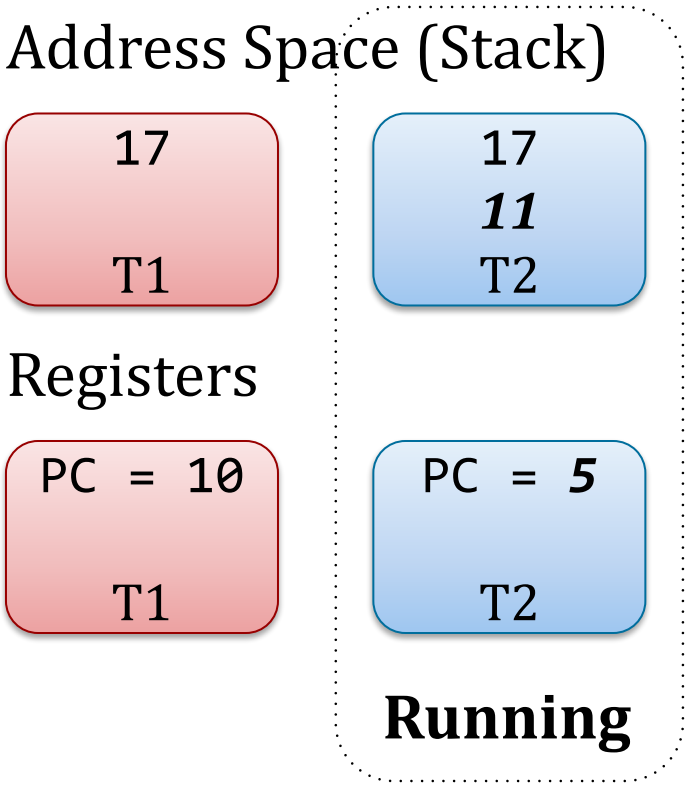


Threads

```
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: }
```



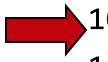




Threads

```
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



Address Space (Data/Heap)

i = 23

Address Space (Stack)

17
T1

17
11
T2

Registers

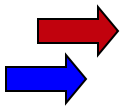
PC = 10
T1

PC = *6*
T2

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: }
```



Address Space (Data/Heap)

i = 23

Address Space (Stack)

17
11
T1

17
11
T2

Registers

PC = ***5***
T1

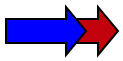
PC = 6
T2

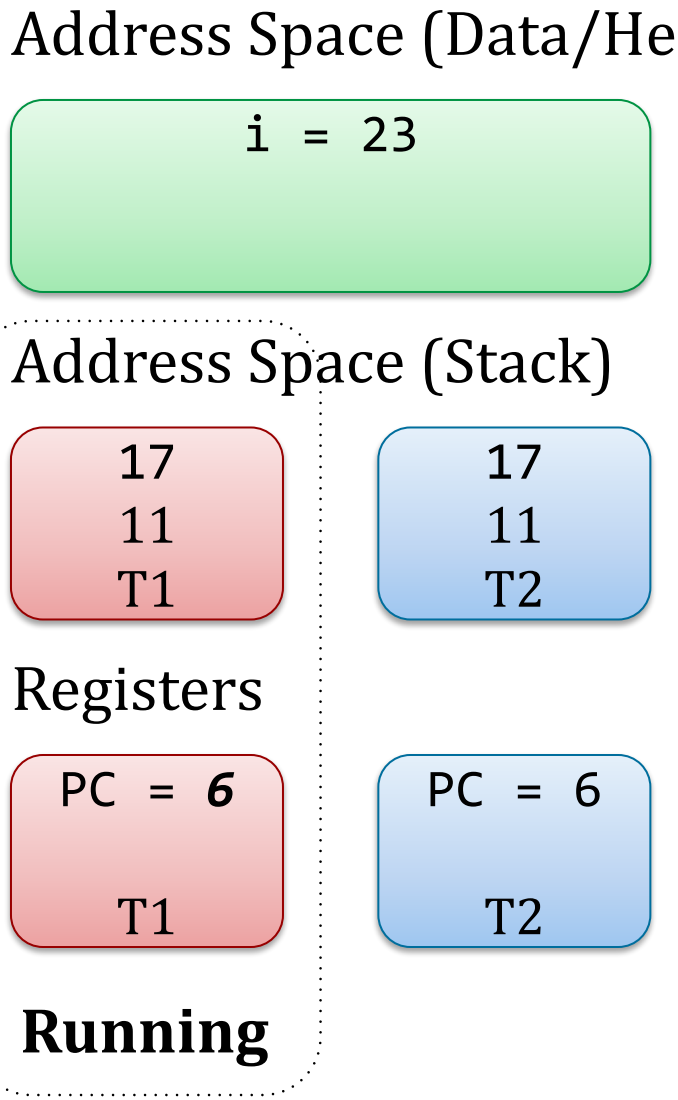
Running

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





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

- `thread_create`
- `thread_exit`
- `thread_join`
- `thread_yield`

(+ 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*)
- No parallelism: only using one CPU

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
- Can exploit multiple cores/CPUs

Linux is a bit wacky...

Linux implements the "clone()" system call.

int clone(int *fn(*void), int clone_flags, …);
CLONE_FS
CLONE_IO
CLONE_SIG
CLONE_VM
…

- passing flags allows callers to specify which process resources are to be duplicated from caller
- omitting flags allows callers to specify which process resources are to be shared with caller

fork() - passes all of these flags so everything gets duplicated in child.
clone(0, CLONE_FS|CLONE_IO|CLONE_SIG|CLONE_VM…);

thread_create() - passes none of the flags so everything is shared in child.
clone(function, 0);

See "man clone" and "man pthread_create" and "man fork" to see differences.

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

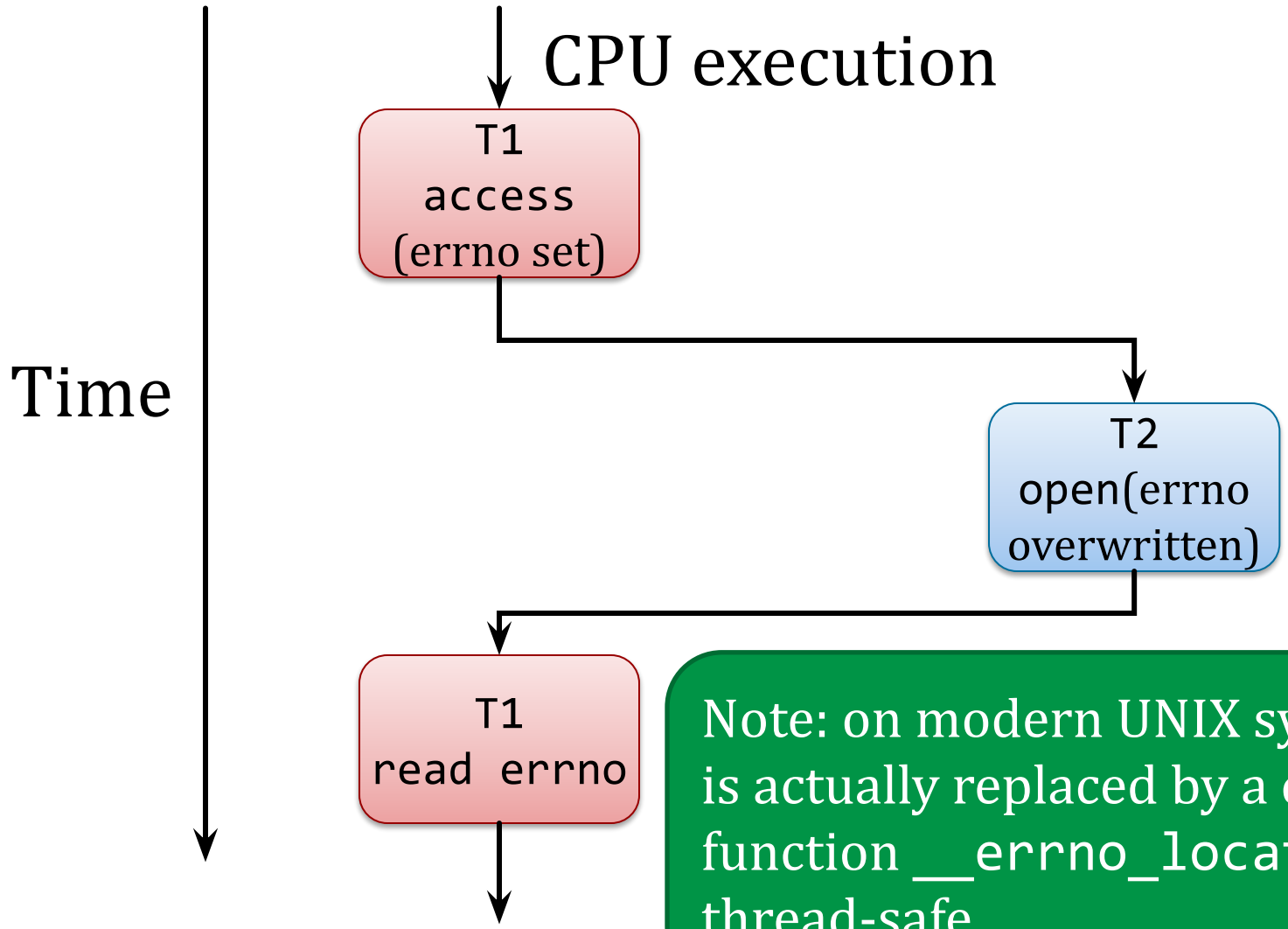
Definition: multiple executions can happen at the same time, e.g.:

- **synchronization (in a couple of lectures)**
- **no side effects, e.g., changes to global variables**

All functions used by threads need to be *reentrant*

What about global variables not protected by locks, e.g., errno?

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