238P Operating Systems, Fall 2018

Final Review
Call Chain of a System Call

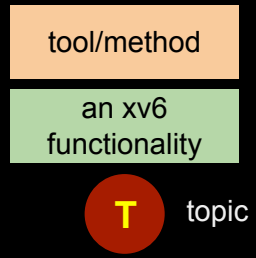
7 December 2018

Aftab Hussain
University of California, Irvine

a visual summary of what
we've done thus far

xv6 is a program

a legend of the diagrams to follow

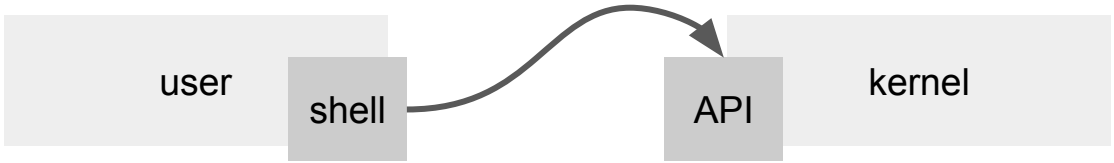


FRONT END

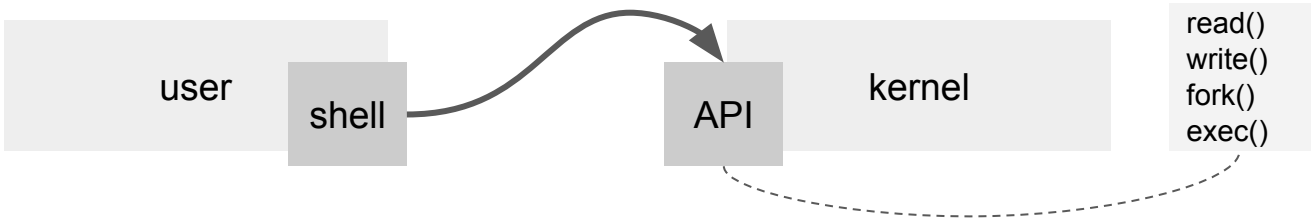
user

kernel

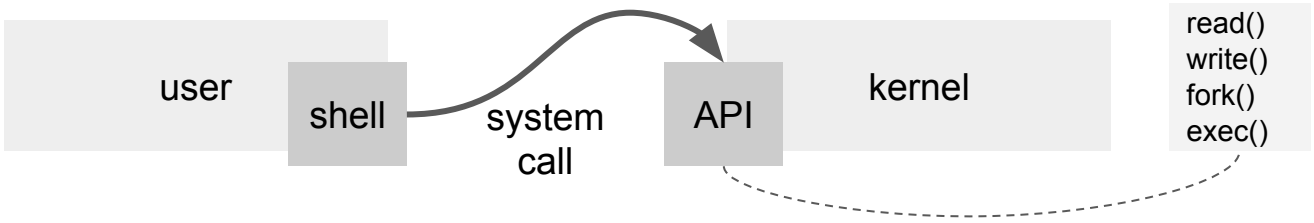
FRONT END

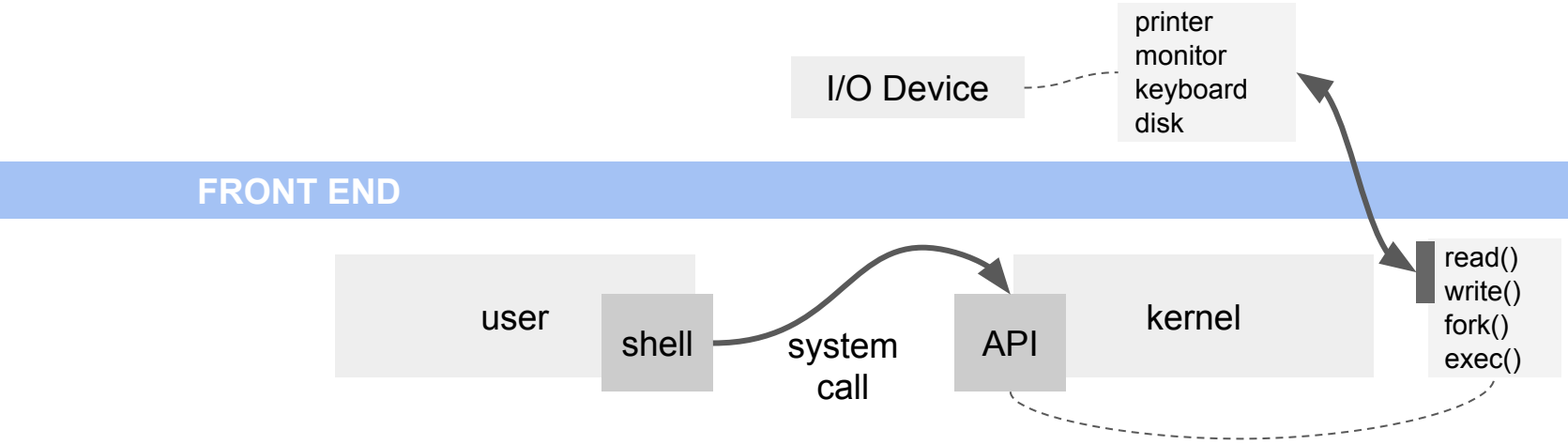


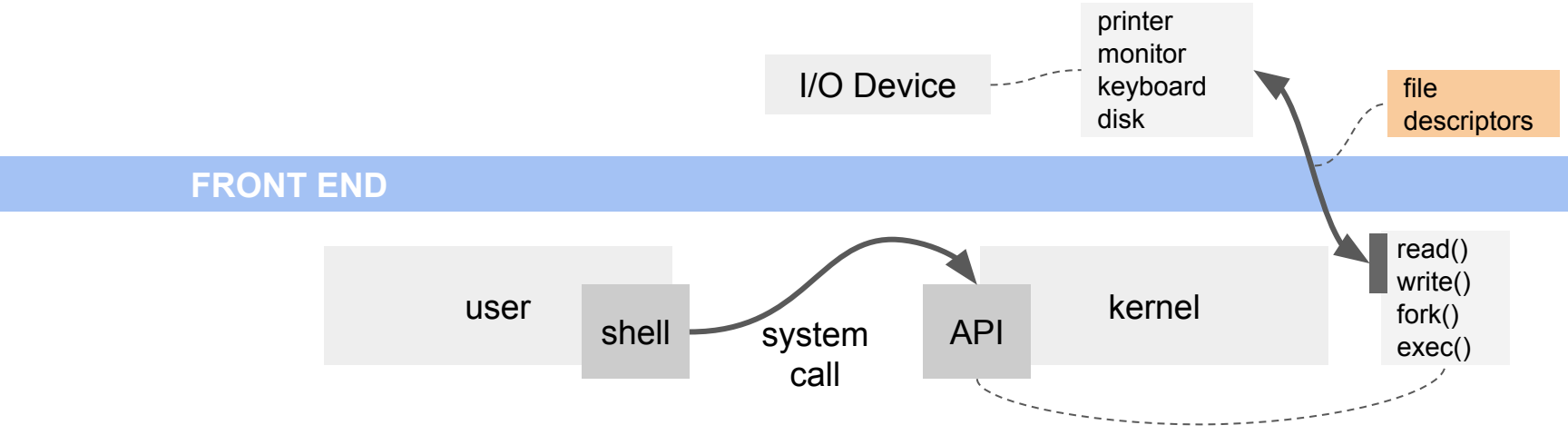
FRONT END

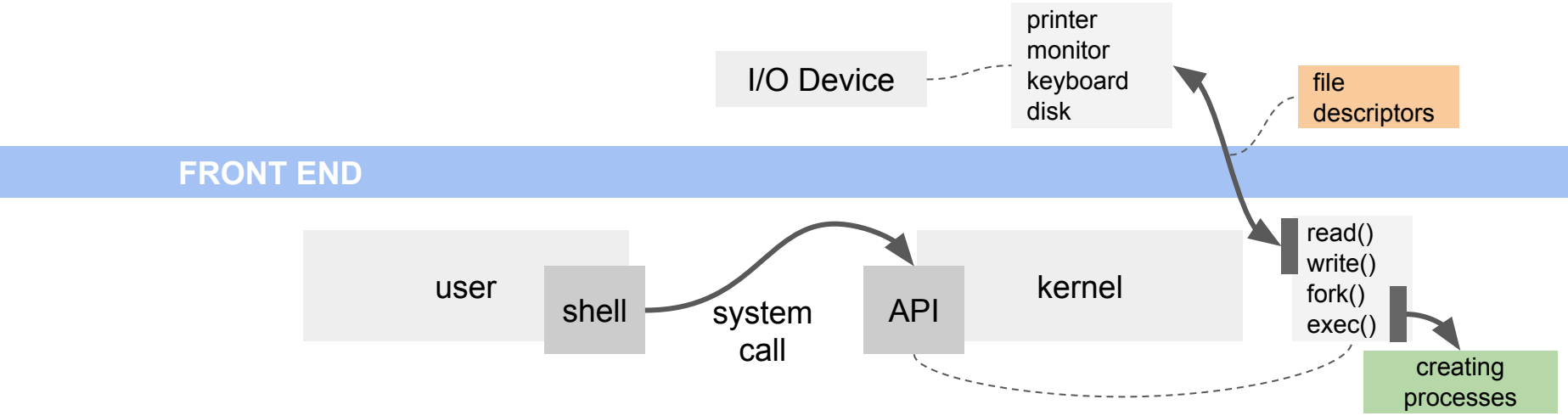


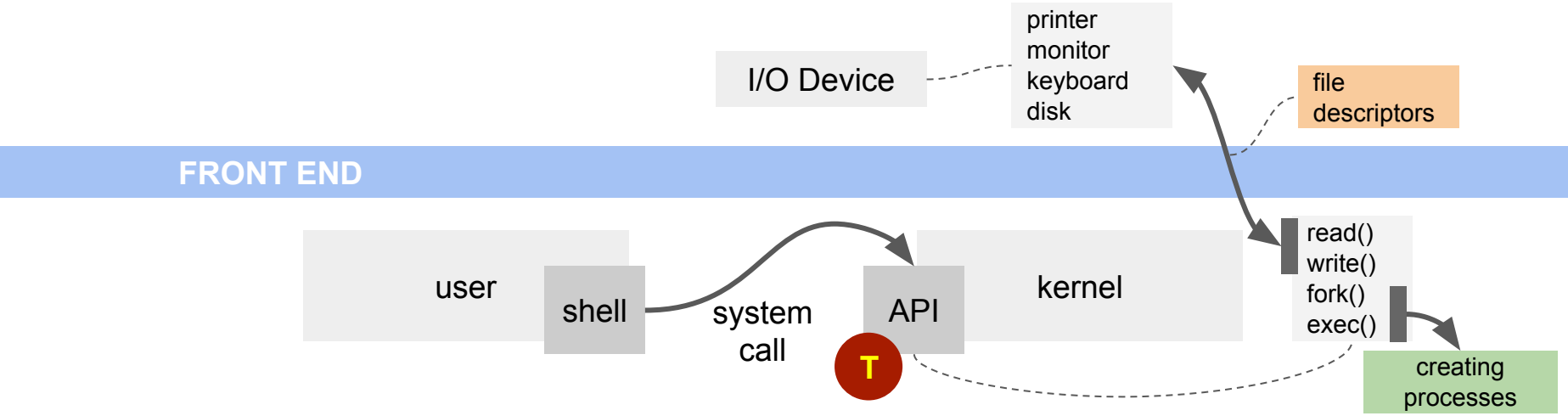
FRONT END



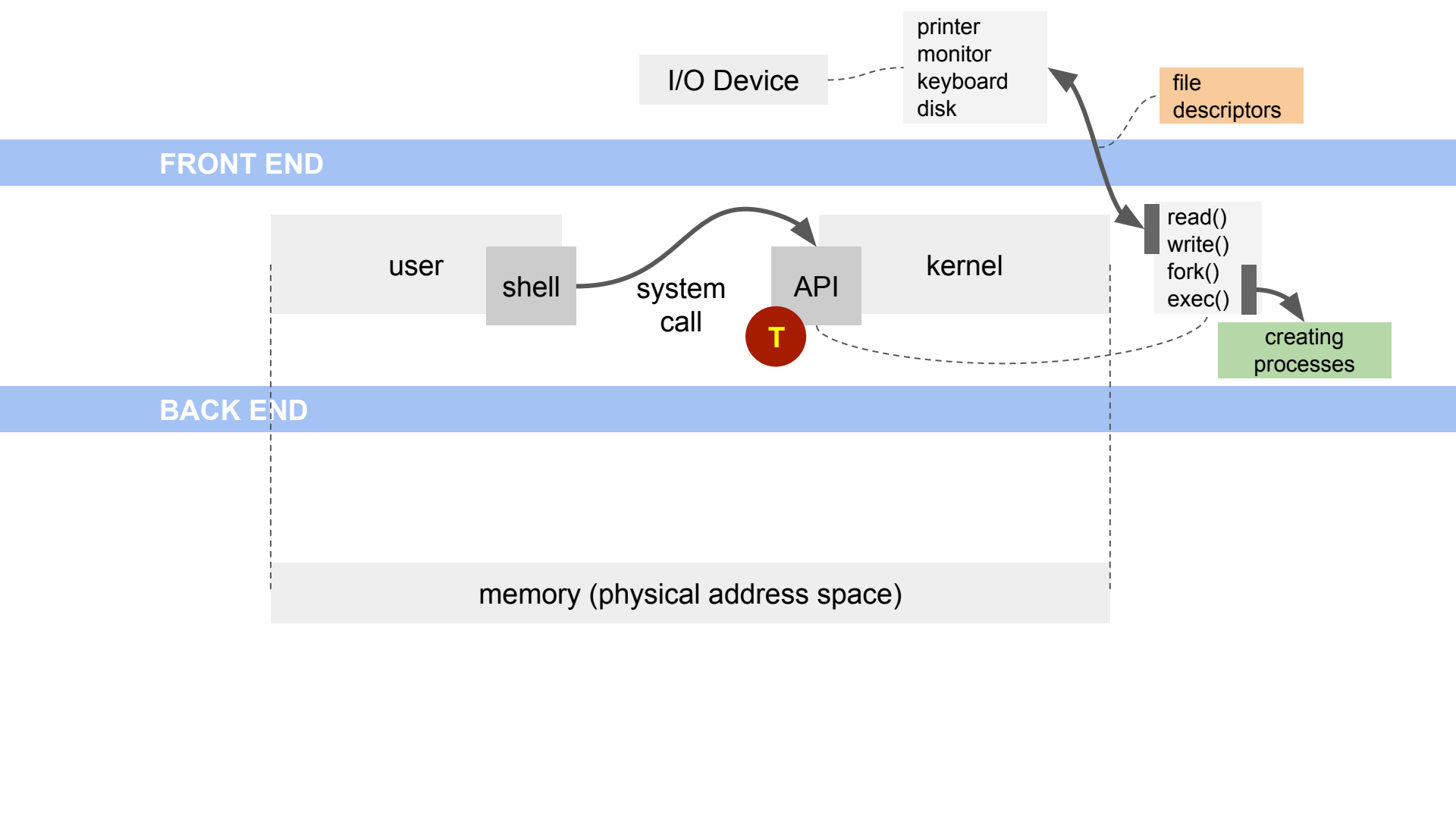

I/O Device
printer
monitor
keyboard
disk
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()

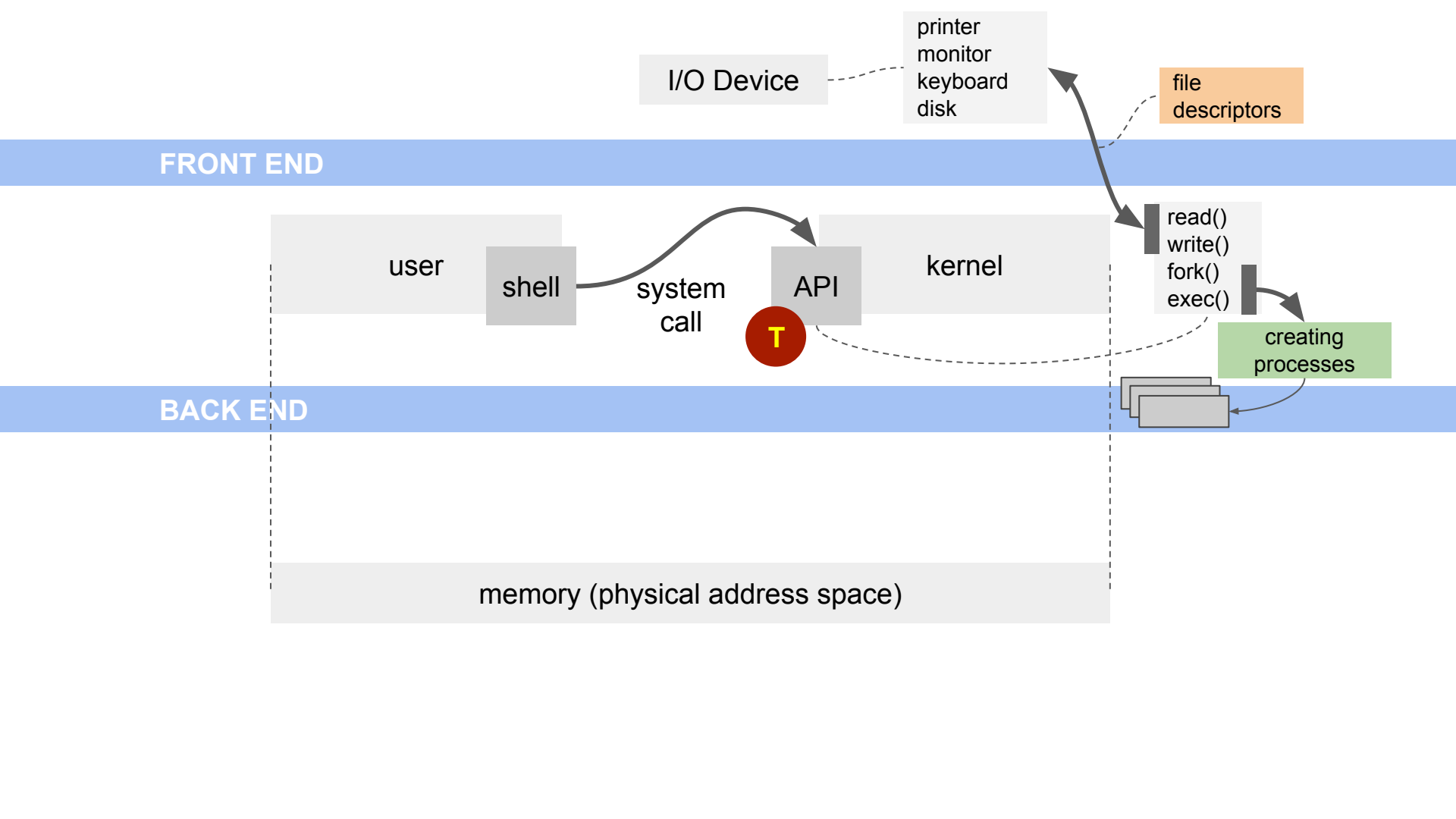

I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
read()
write()
fork()
exec()
user
shell
system
call
API
kernel

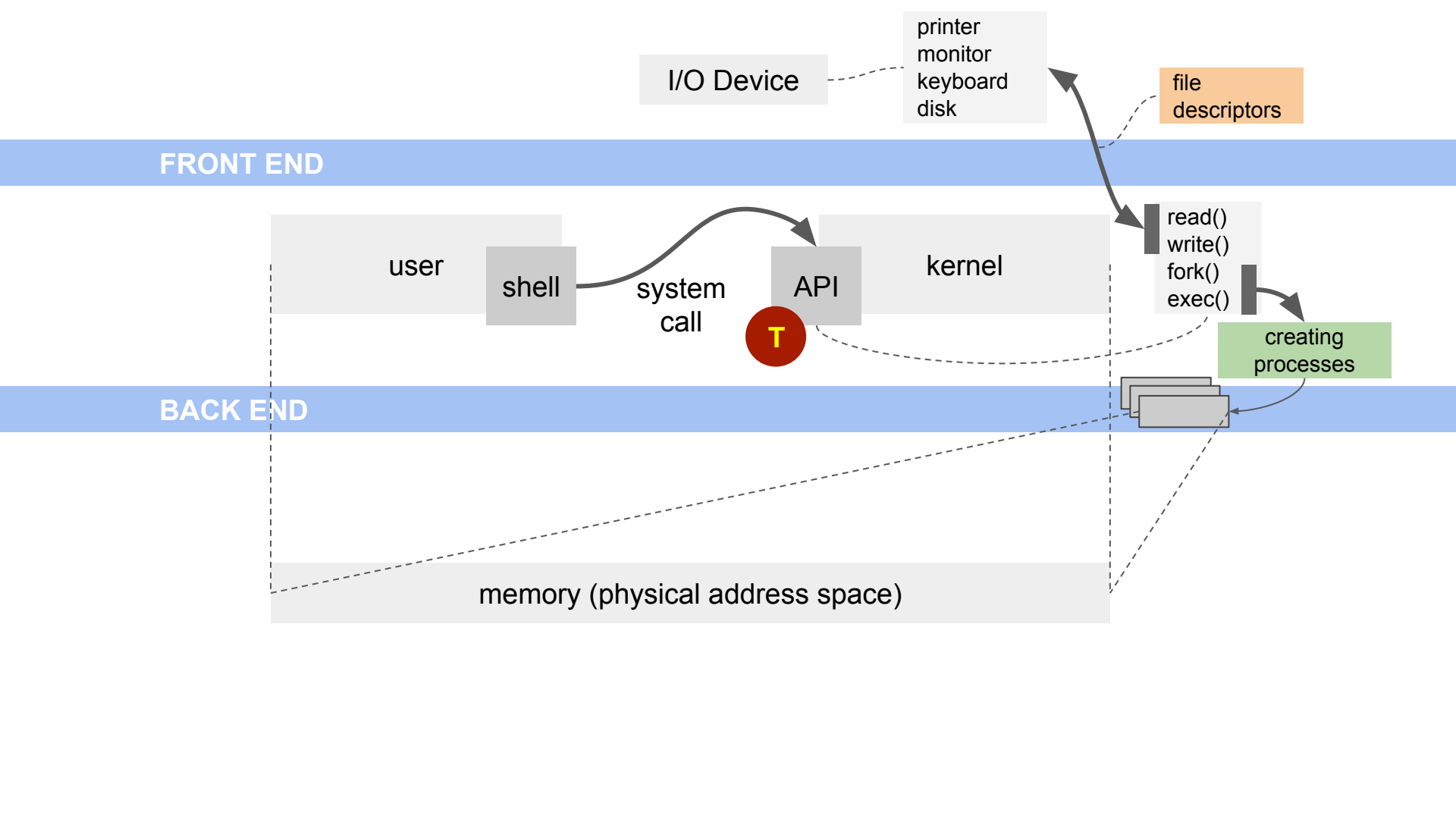

I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes

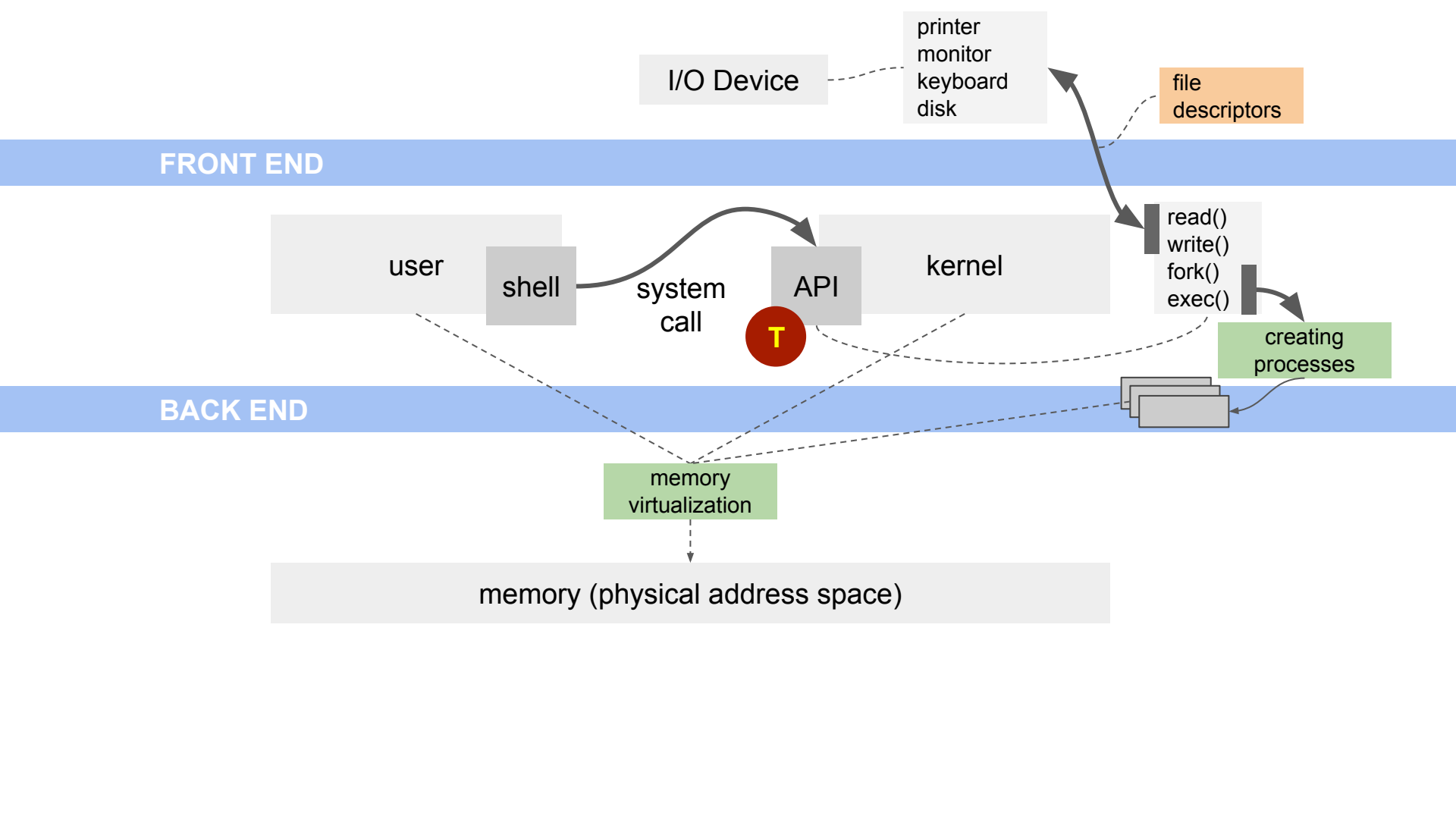

I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
T

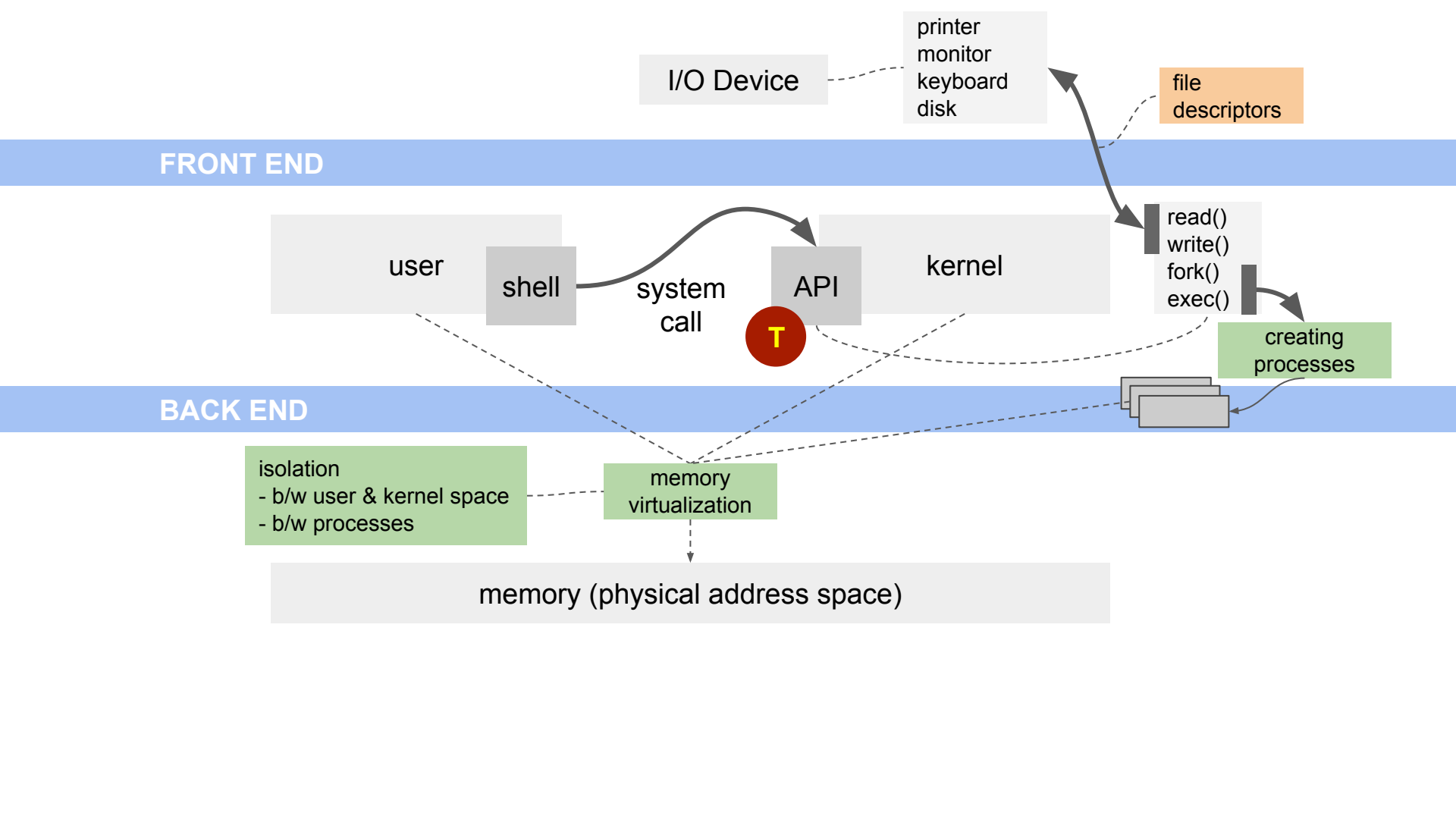
Any running program -- i.e., *a process* -- must be in memory, so the CPU can run it.

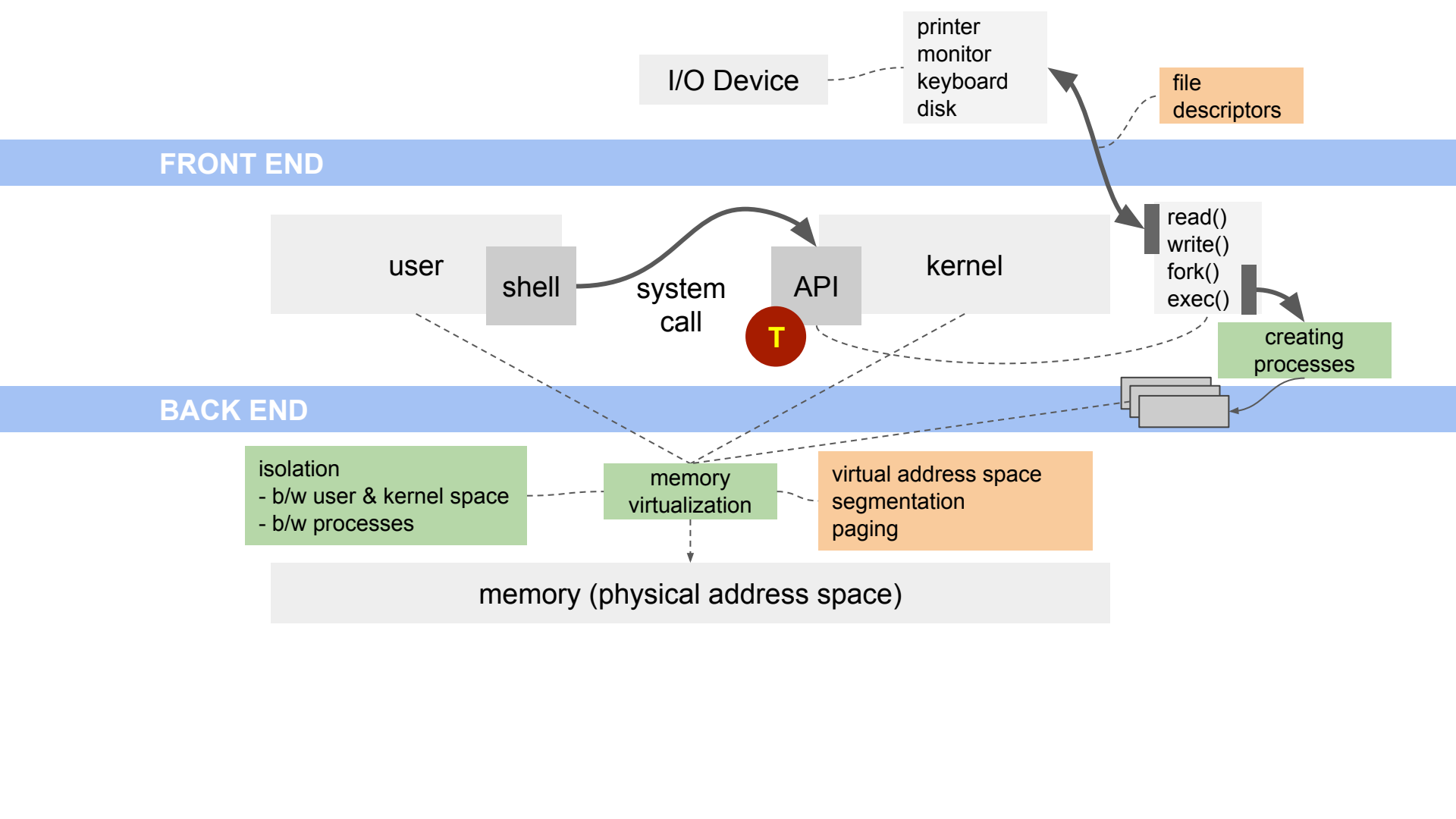

I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
T
BACK END
memory (physical address space)

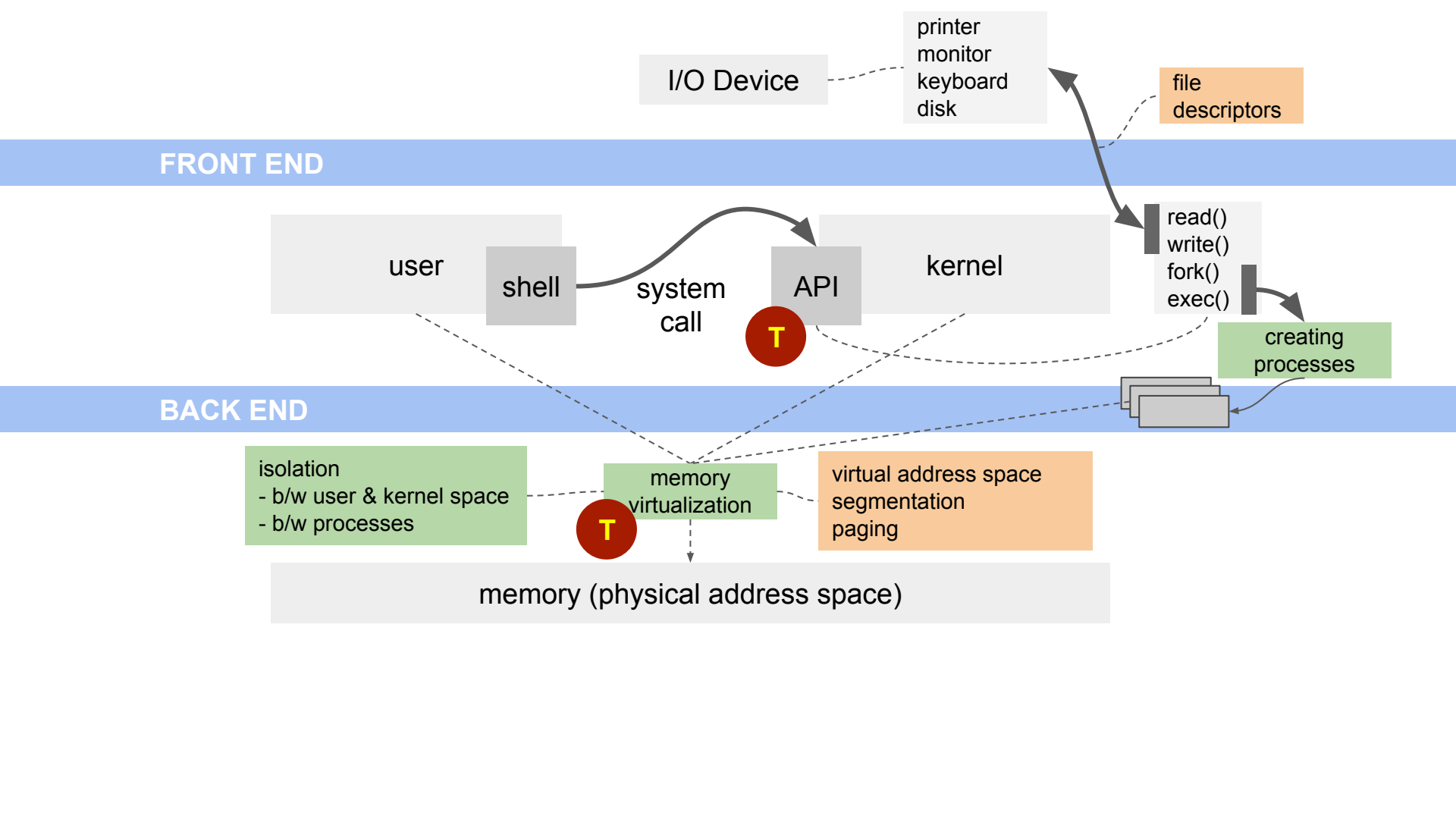

I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
T
BACK END
memory (physical address space)


I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
T
BACK END
memory (physical address space)

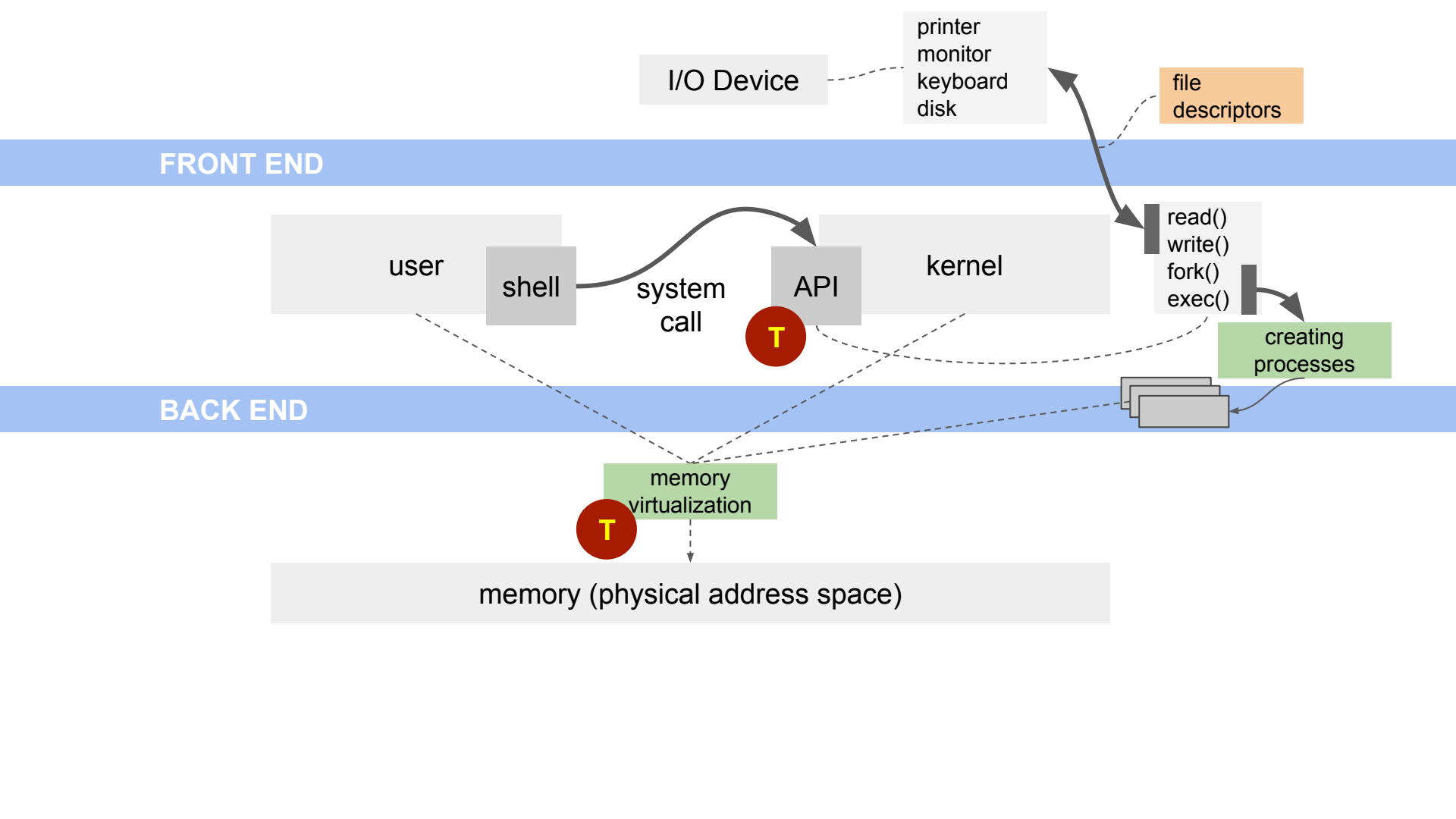

I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
T
creating
processes
BACK END
memory
virtualization
memory (physical address space)

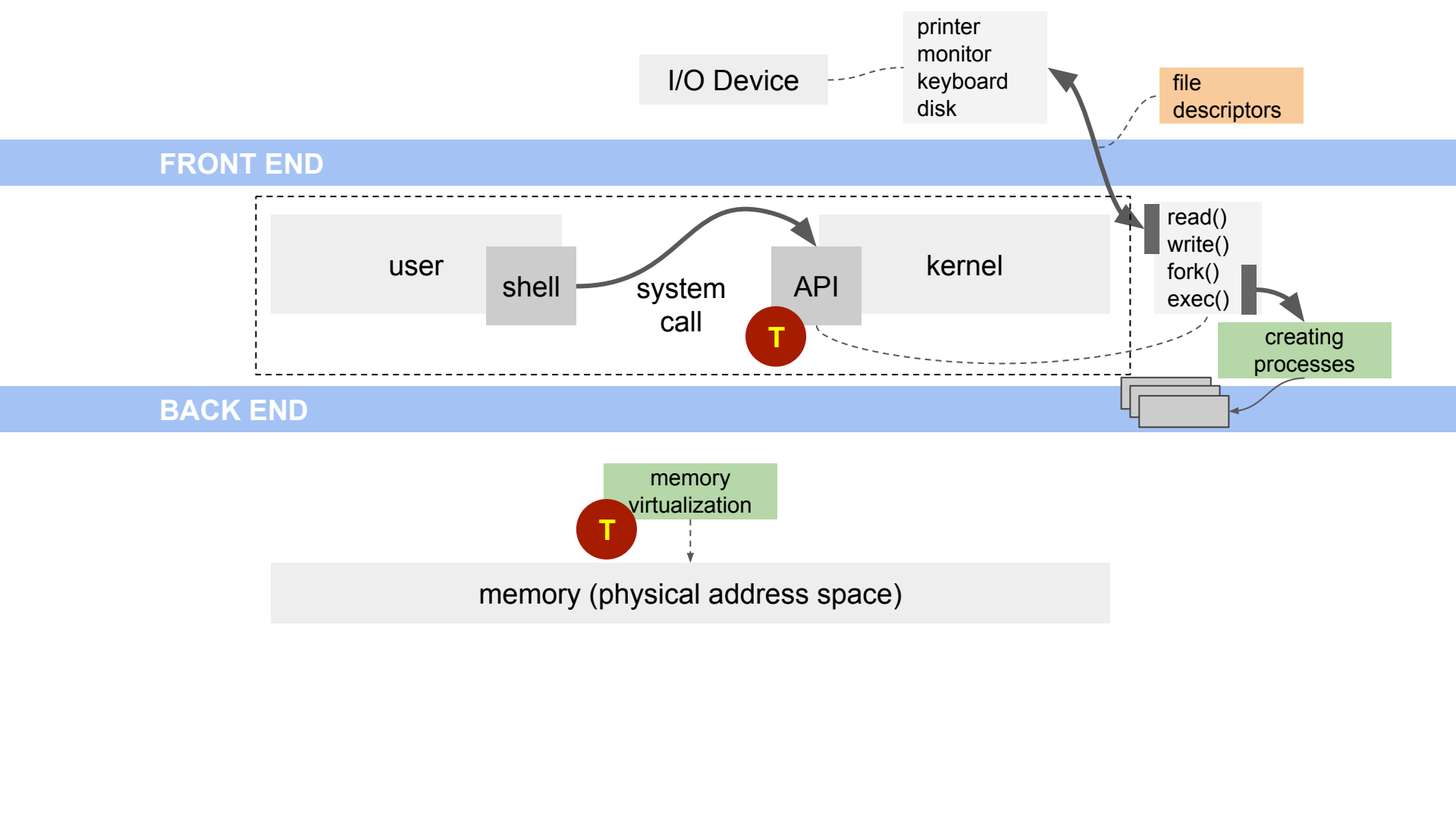

I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
T
creating
processes
BACK END
isolation
- b/w user & kernel space
- b/w processes
memory
virtualization
memory (physical address space)

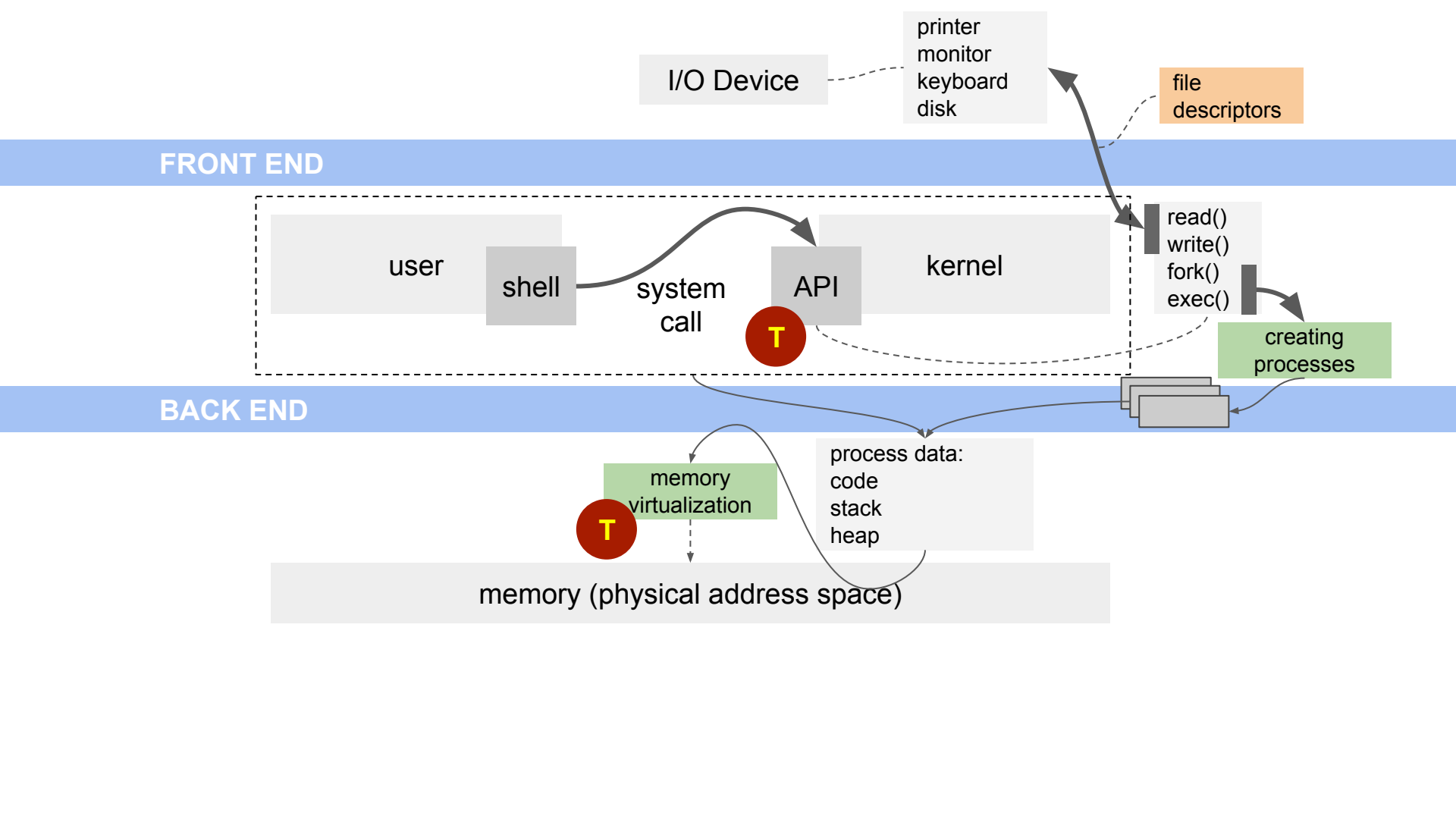

I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
T
creating
processes
BACK END
isolation
- b/w user & kernel space
- b/w processes
memory
virtualization
virtual address space
segmentation
paging
memory (physical address space)

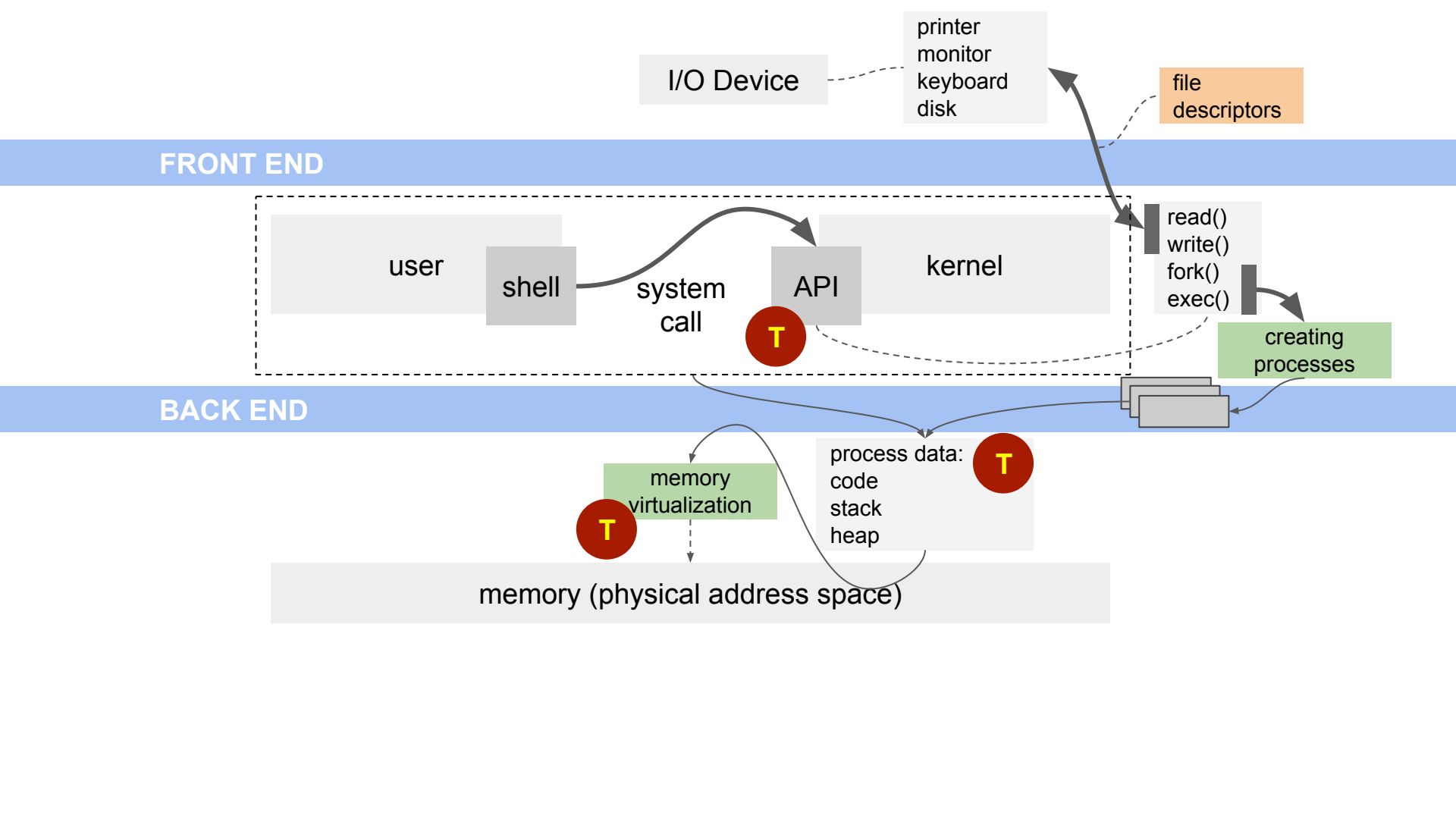

I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
T
BACK END
isolation
- b/w user & kernel space
- b/w processes
memory
virtualization
T
virtual address space
segmentation
paging
memory (physical address space)

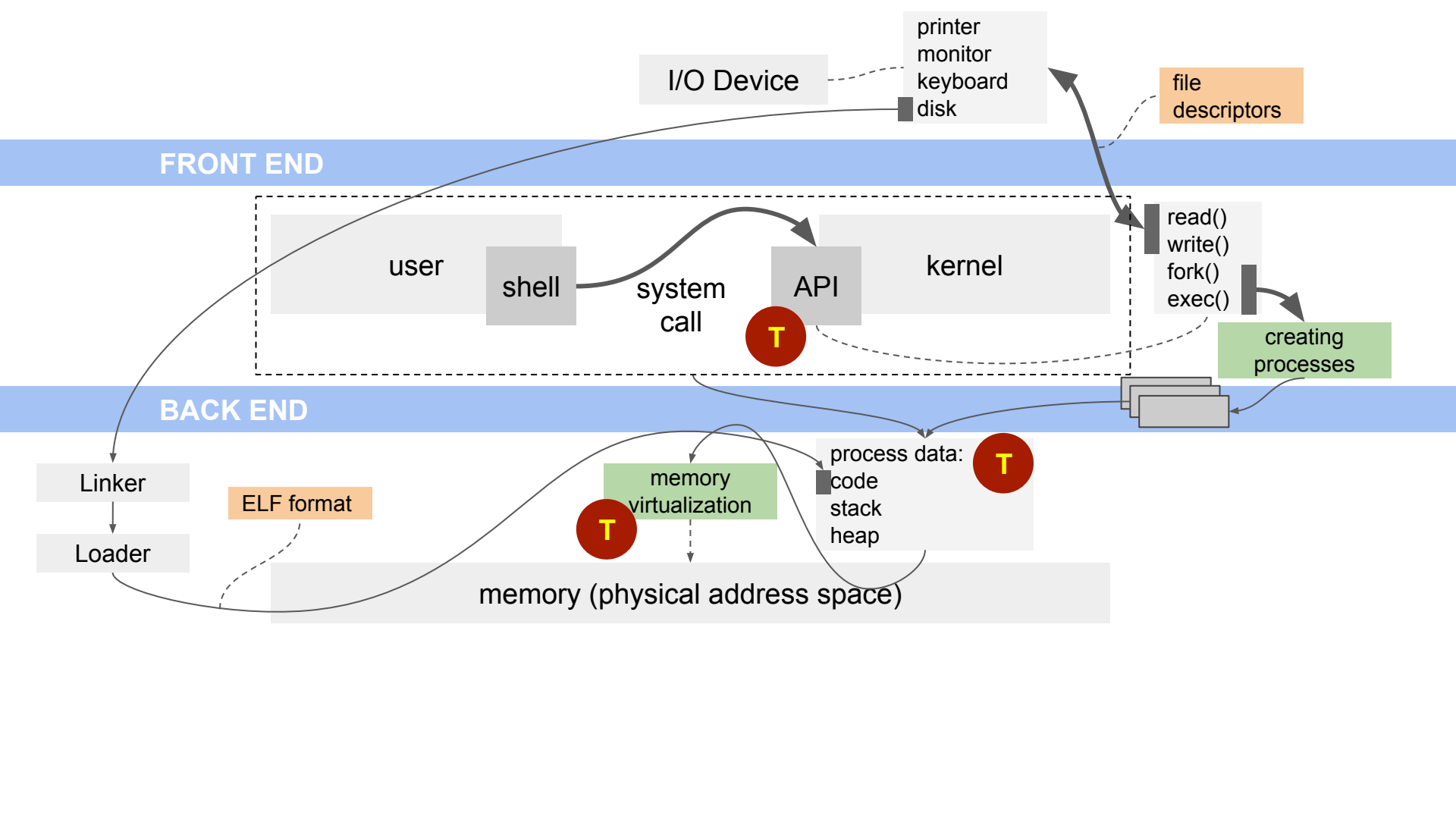
So what exactly must be in memory?

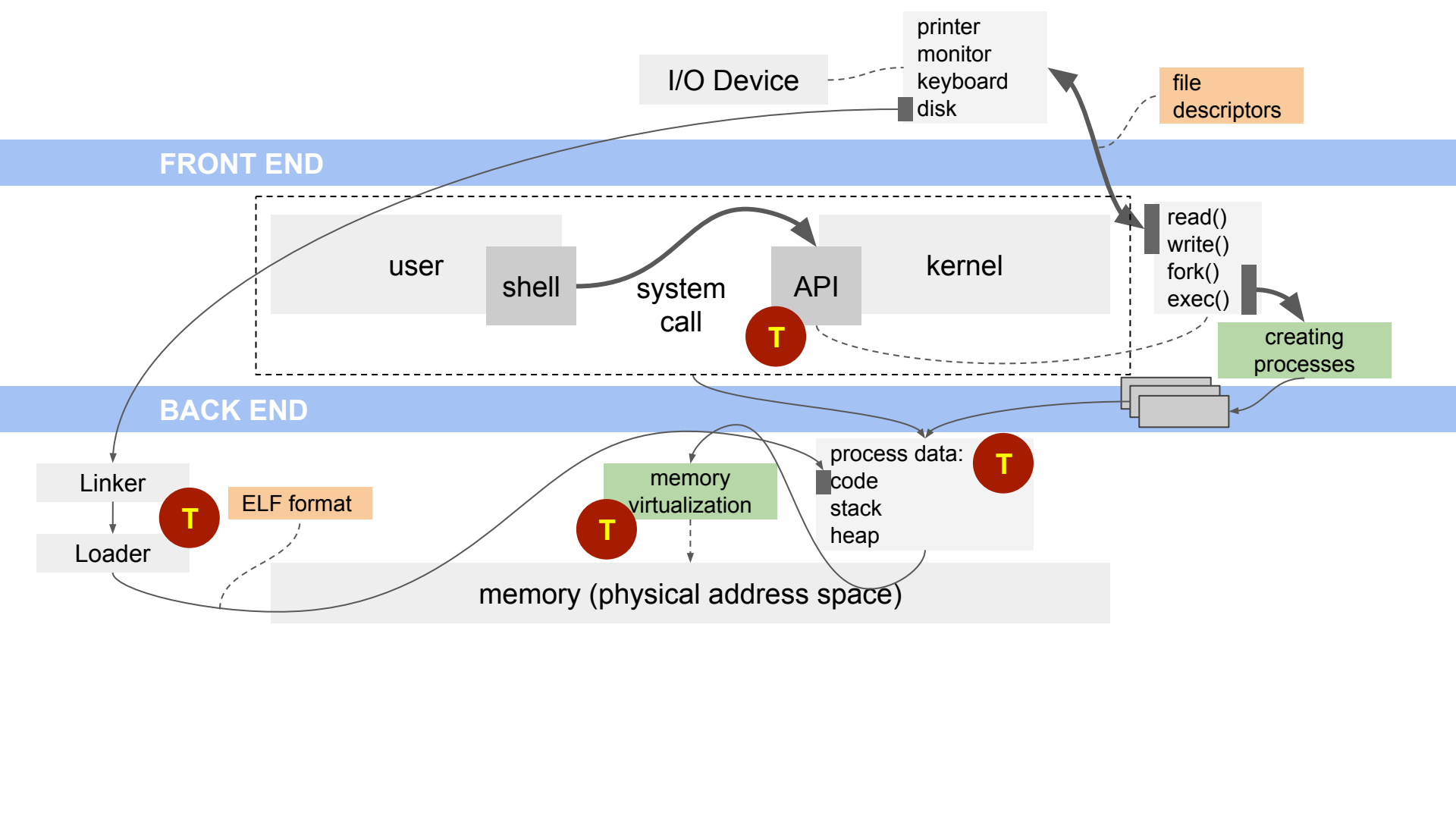

I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
T
BACK END
memory
virtualization
T
memory (physical address space)


I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
T
BACK END
memory
virtualization
T
memory (physical address space)

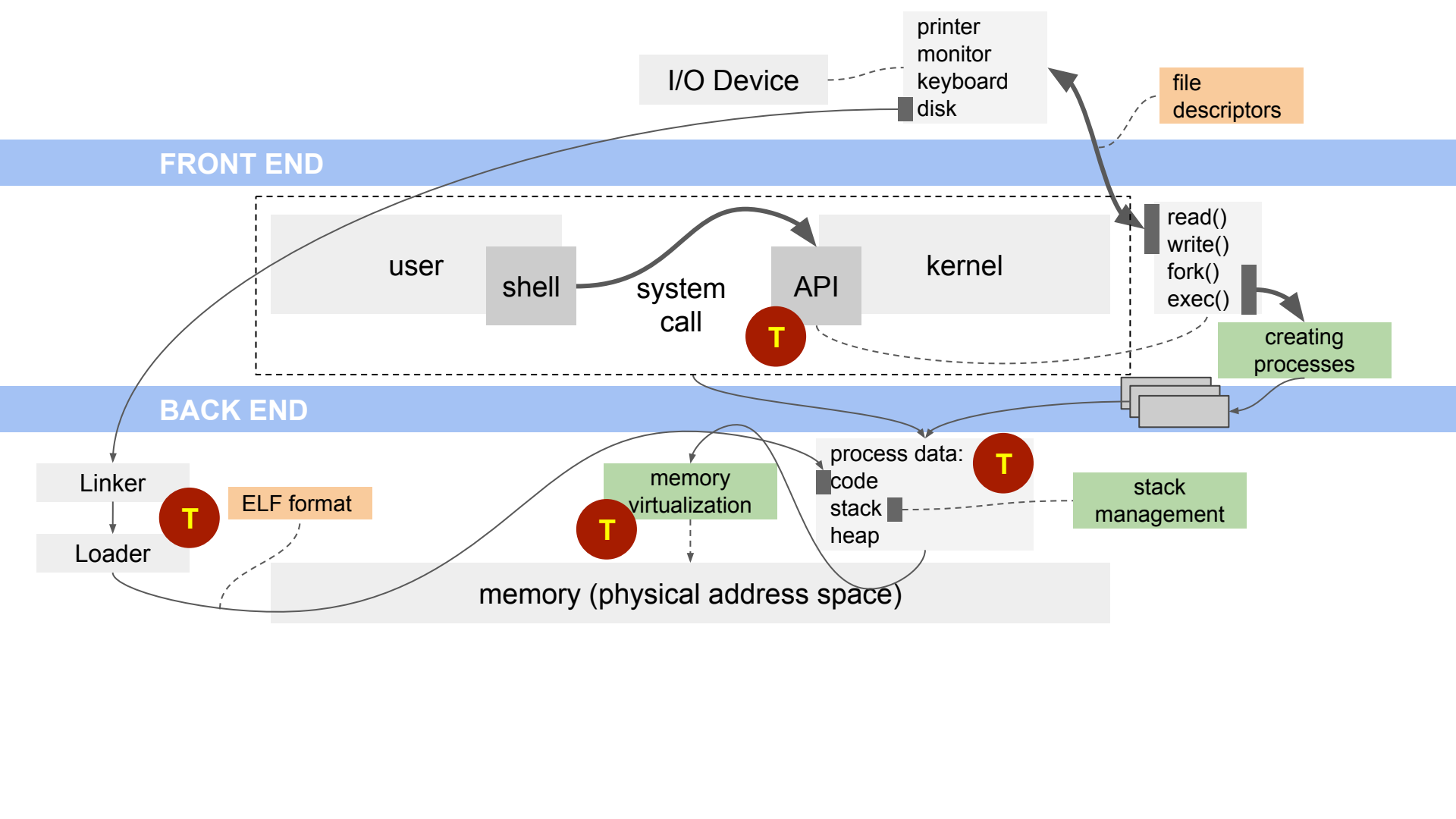

I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
T
BACK END
memory
virtualization
T
process data:
code
stack
heap
memory (physical address space)

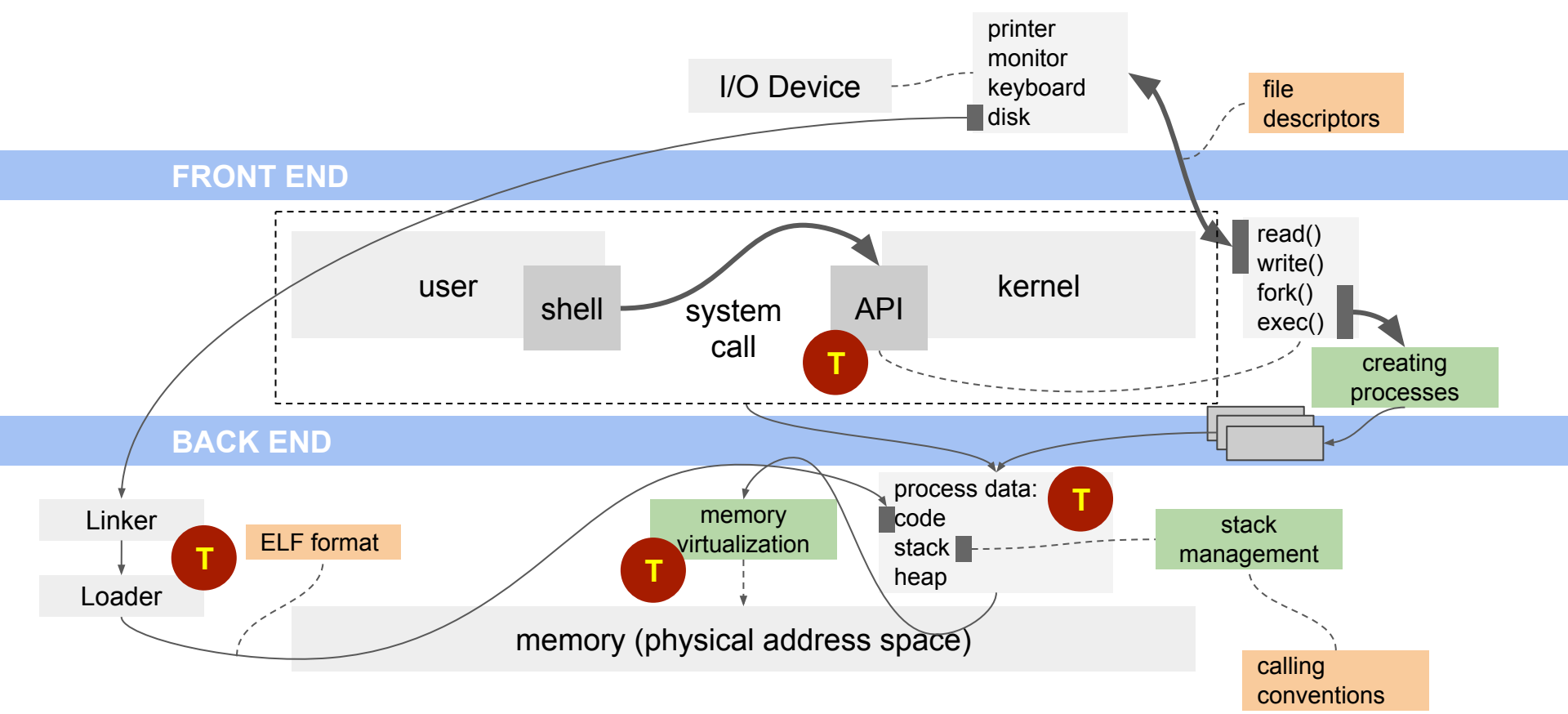

I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
T
BACK END
memory
virtualization
T
process data:
code
stack
heap
T
memory (physical address space)

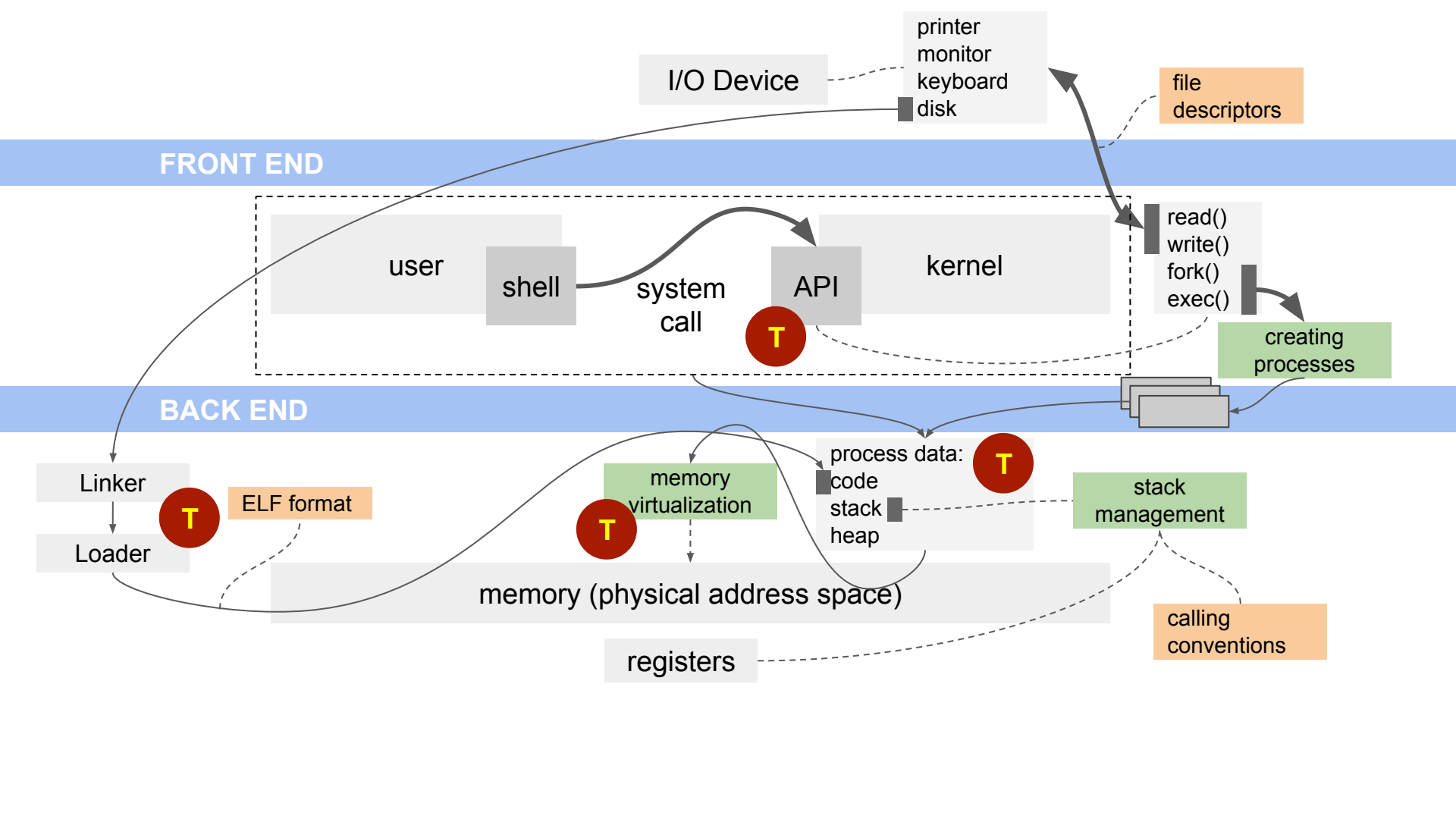

I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
T
BACK END
Linker
Loader
ELF format
memory
virtualization
process data:
code
stack
heap
memory (physical address space)

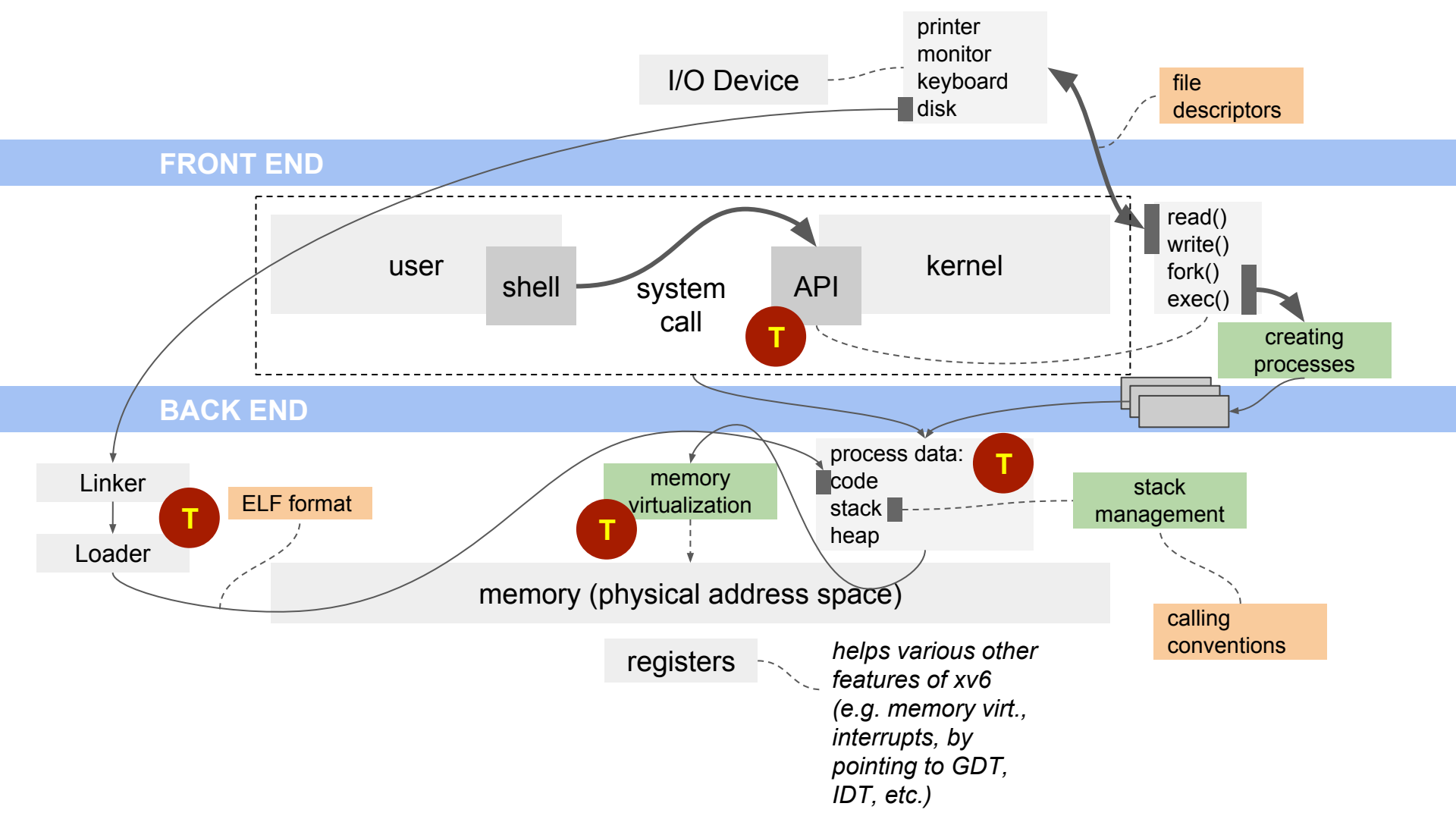

I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
T
BACK END
Linker
Loader
T
ELF format
memory
virtualization
T
process data:
code
stack
heap
T
memory (physical address space)

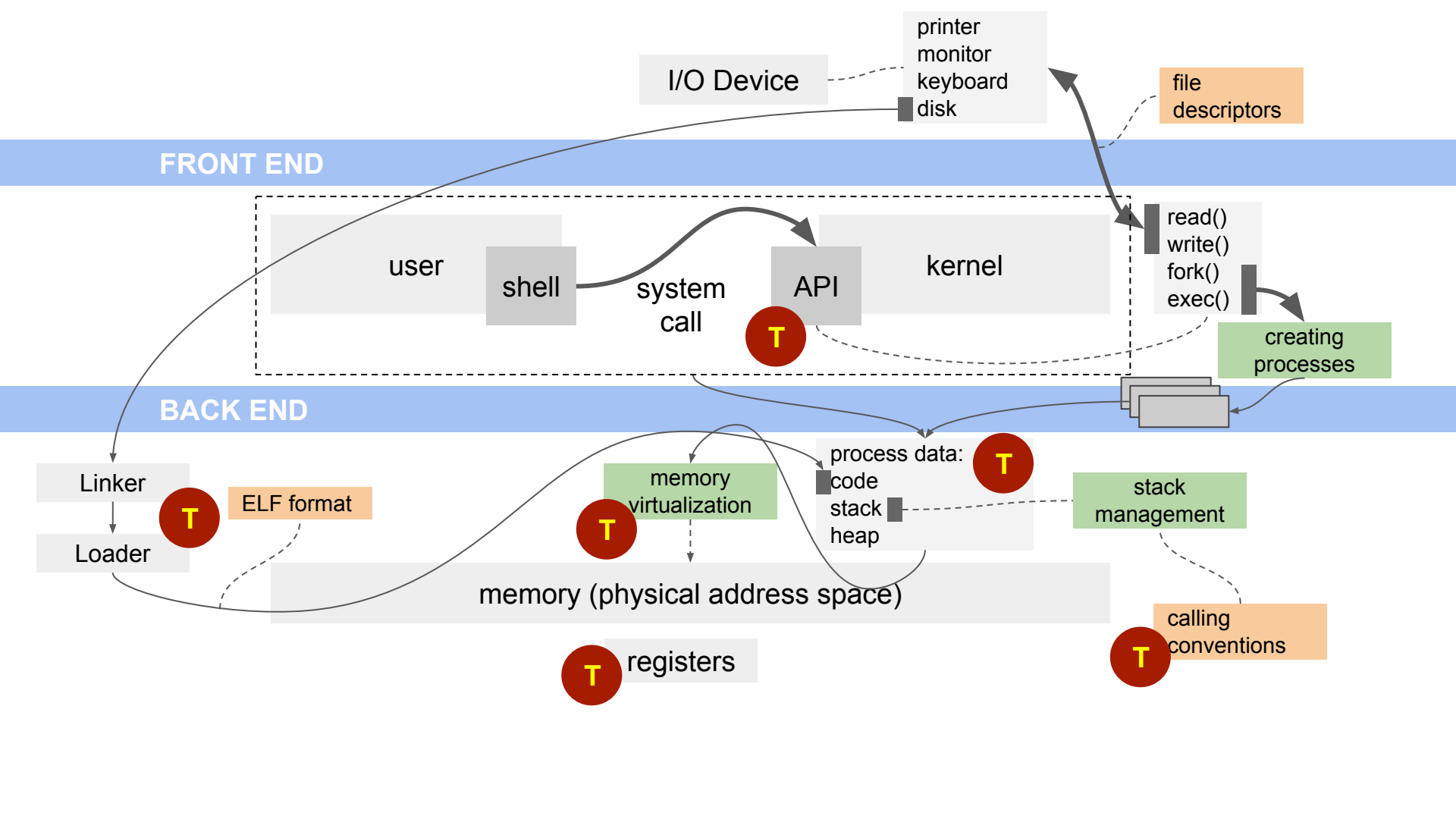
Need to handle dynamic info.


I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
BACK END
Linker
Loader
ELF format
memory
virtualization
process data:
code
stack
heap
stack
management
memory (physical address space)

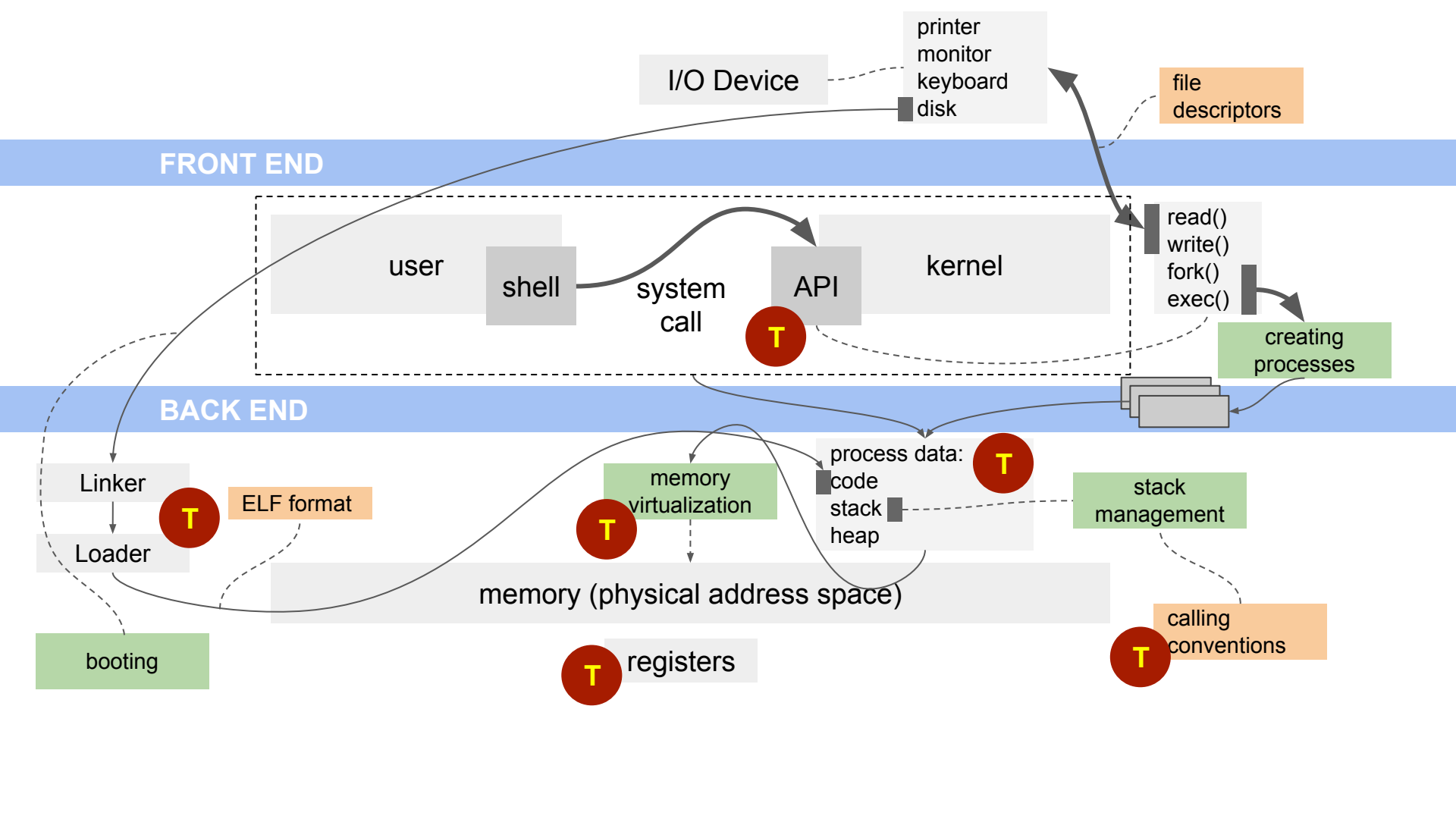

I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
T
BACK END
Linker
Loader
T
ELF format
memory
virtualization
T
process data:
code
stack
heap
T
stack
management
memory (physical address space)
calling
conventions

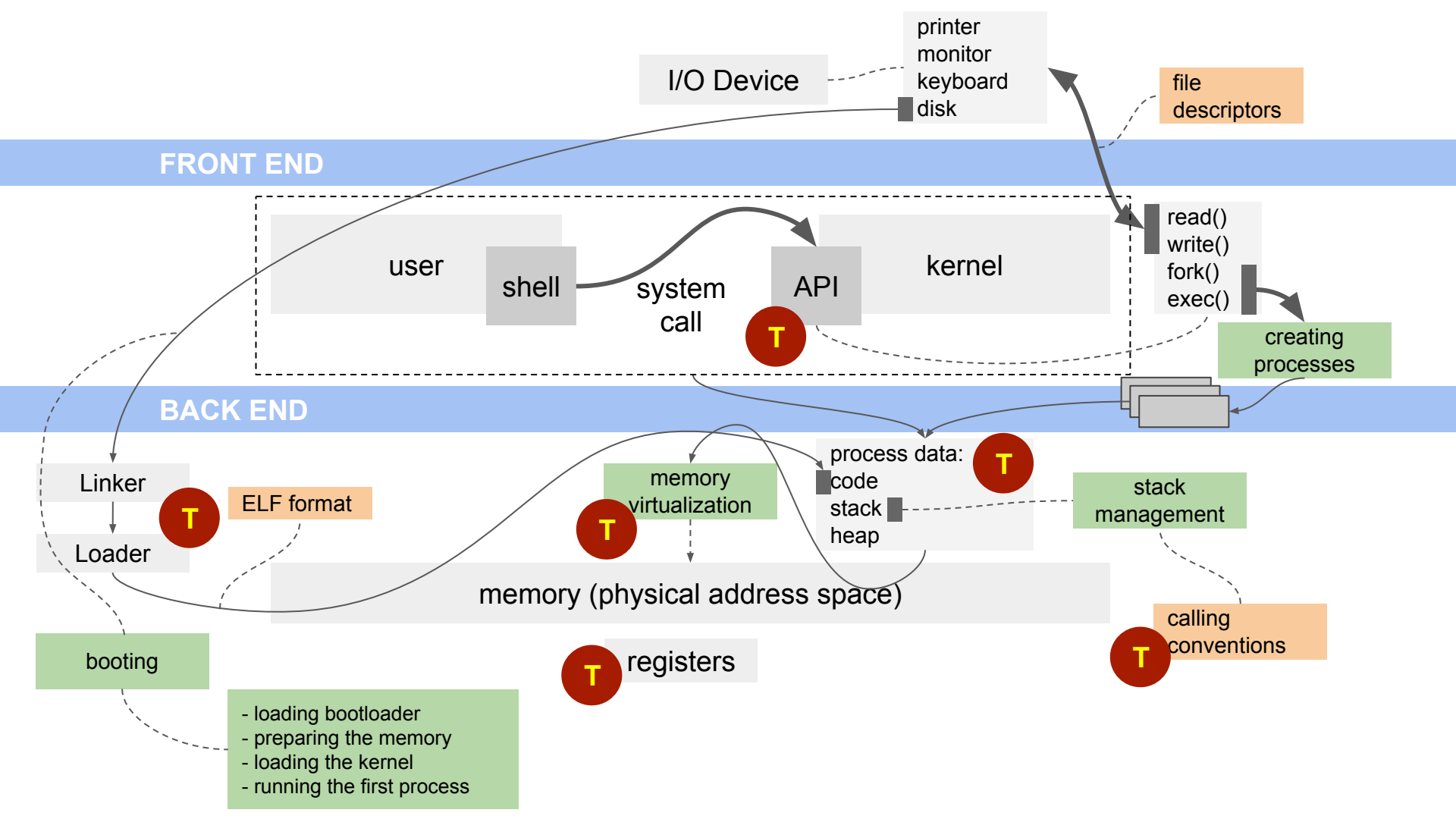

I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
T
BACK END
Linker
Loader
T
ELF format
memory
virtualization
T
process data:
code
stack
heap
T
stack
management
memory (physical address space)
registers
calling
conventions

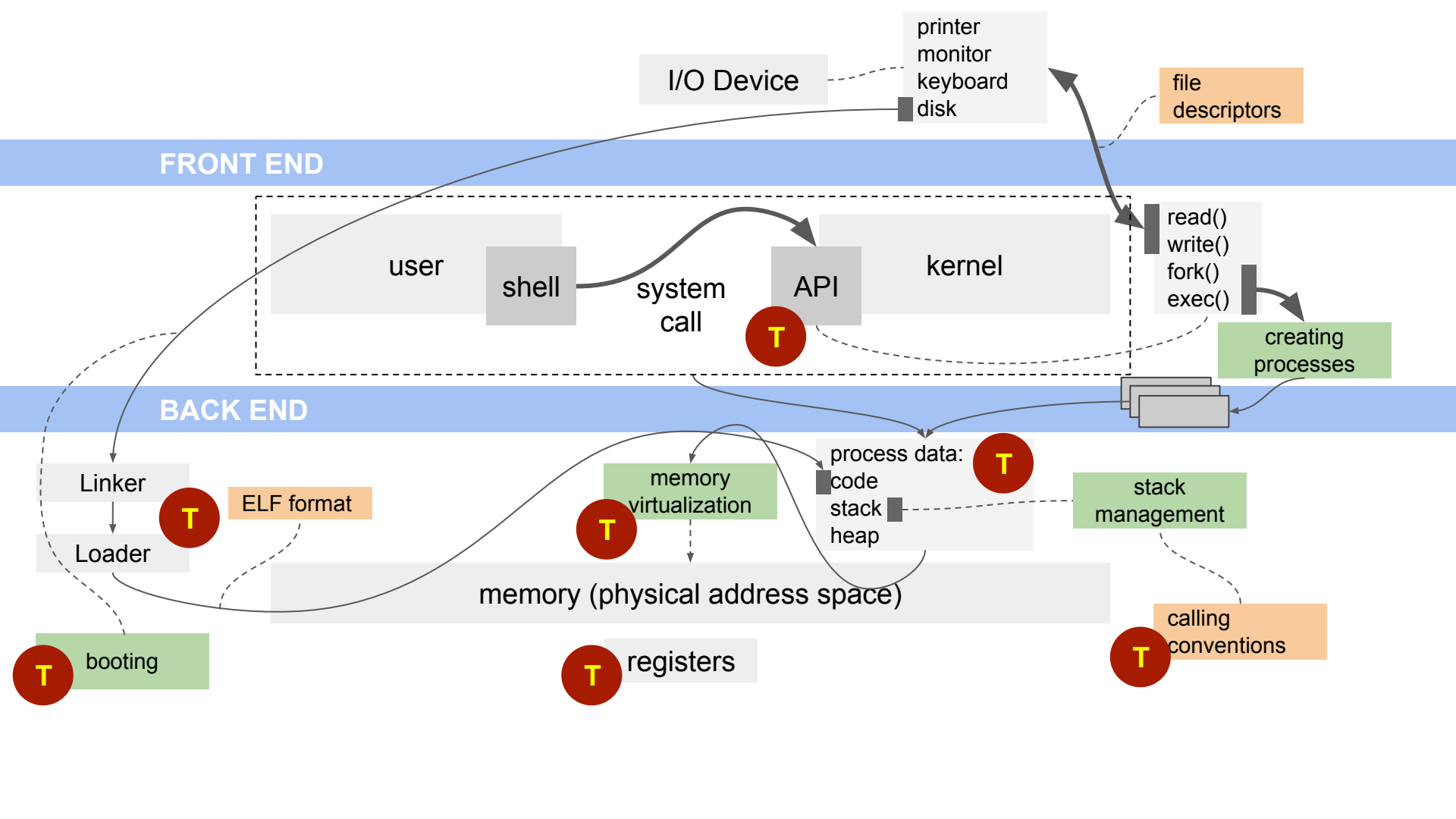

I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
T
creating
processes
BACK END
Linker
Loader
T
ELF format
memory
virtualization
T
process data:
code
stack
heap
T
stack
management
memory (physical address space)
calling
conventions
registers
helps various other features of xv6 (e.g. memory virt., interrupts, by pointing to GDT, IDT, etc.)


I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
T
BACK END
Linker
Loader
ELF format
memory
virtualization
process data:
code
stack
heap
stack
management
memory (physical address space)
calling
conventions
registers

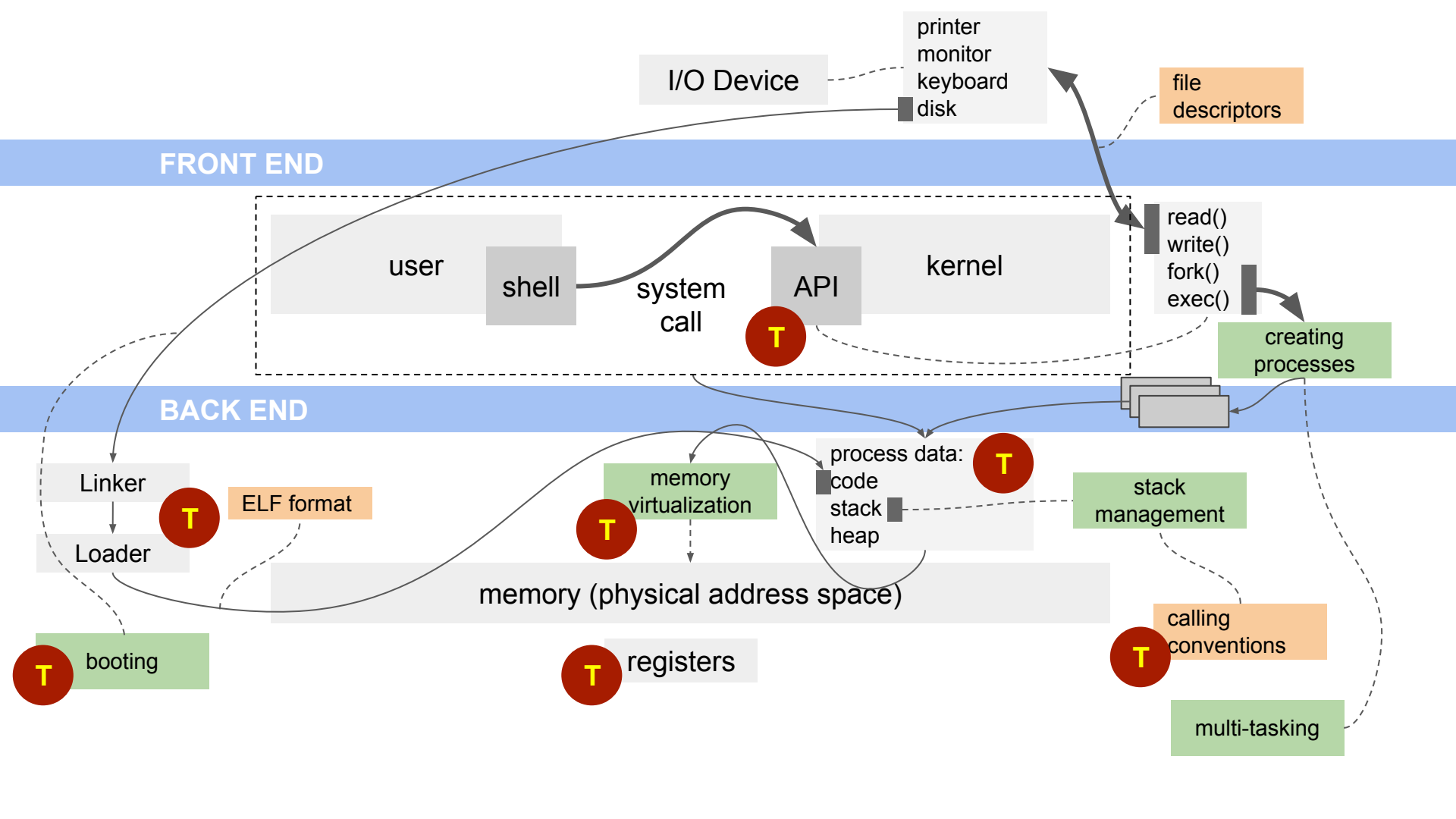
Did we forget to load the OS -- or kernel -- itself?

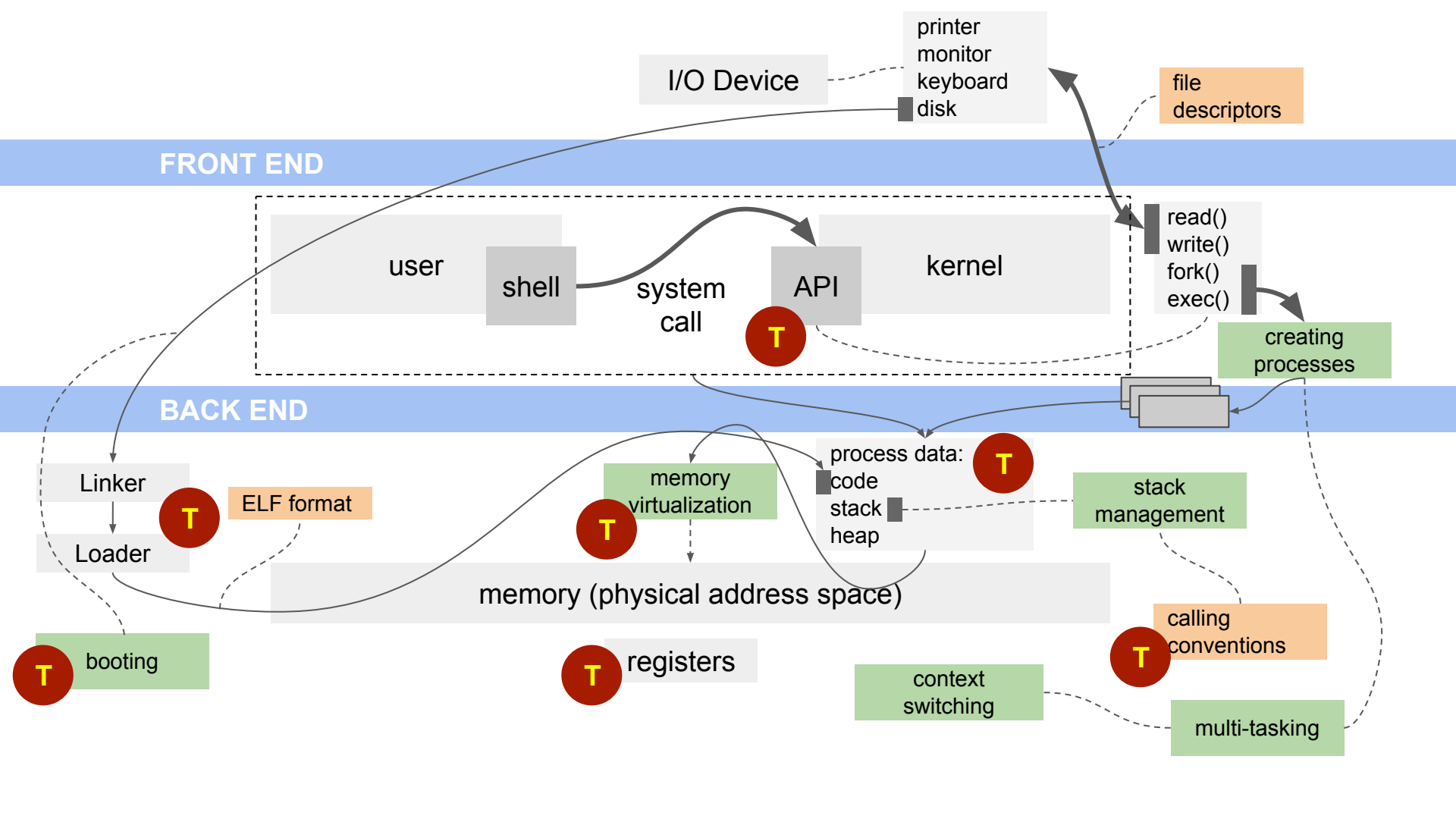

I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
BACK END
Linker
ELF format
Loader
booting
memory
virtualization
process data:
code
stack
heap
stack
management
memory (physical address space)
calling
conventions
registers
T

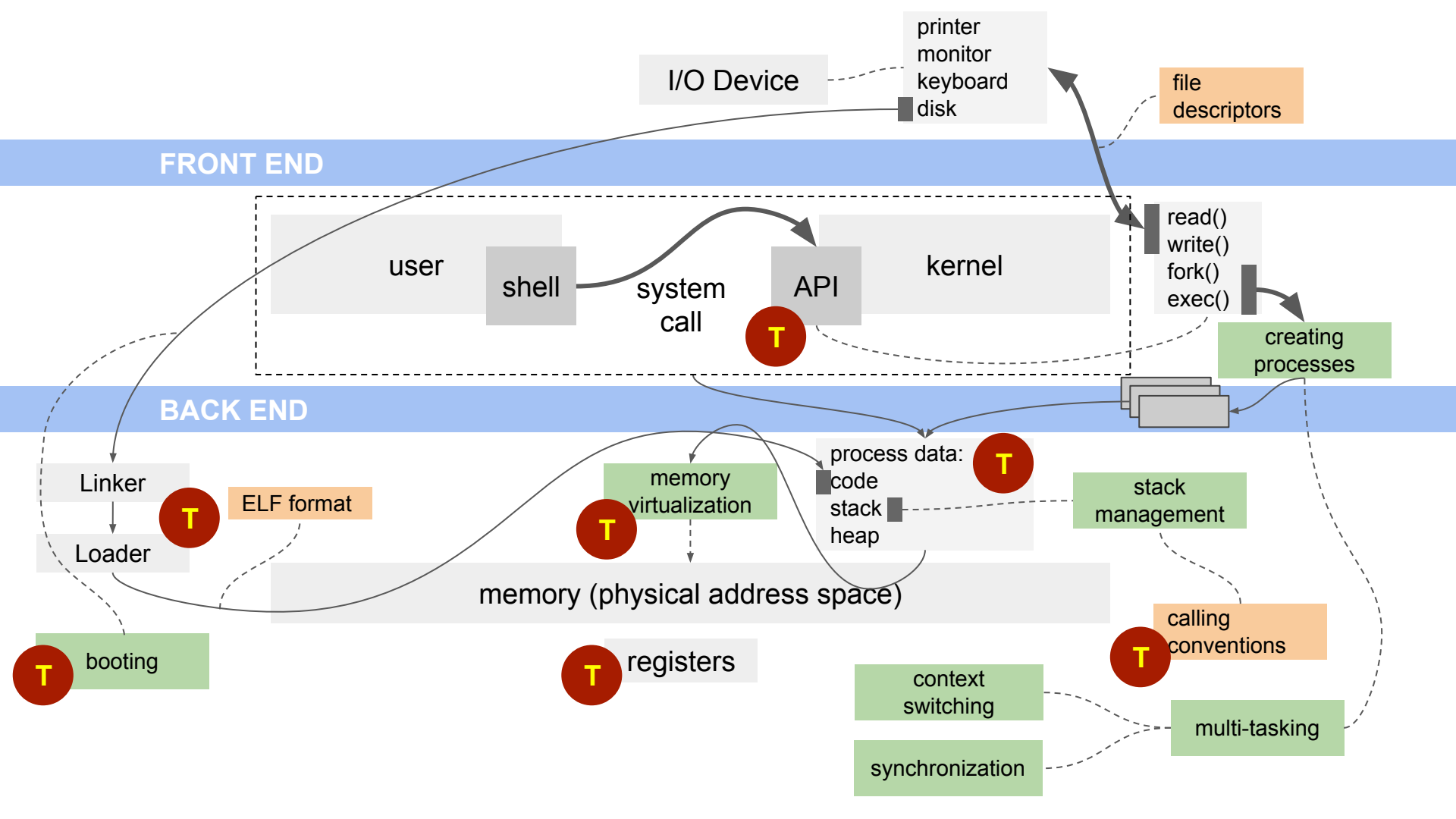

I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
BACK END
Linker
ELF format
Loader
memory
virtualization
process data:
code
stack
heap
stack
management
memory (physical address space)
booting
registers
calling
conventions
- loading bootloader
- preparing the memory
- loading the kernel
- running the first process

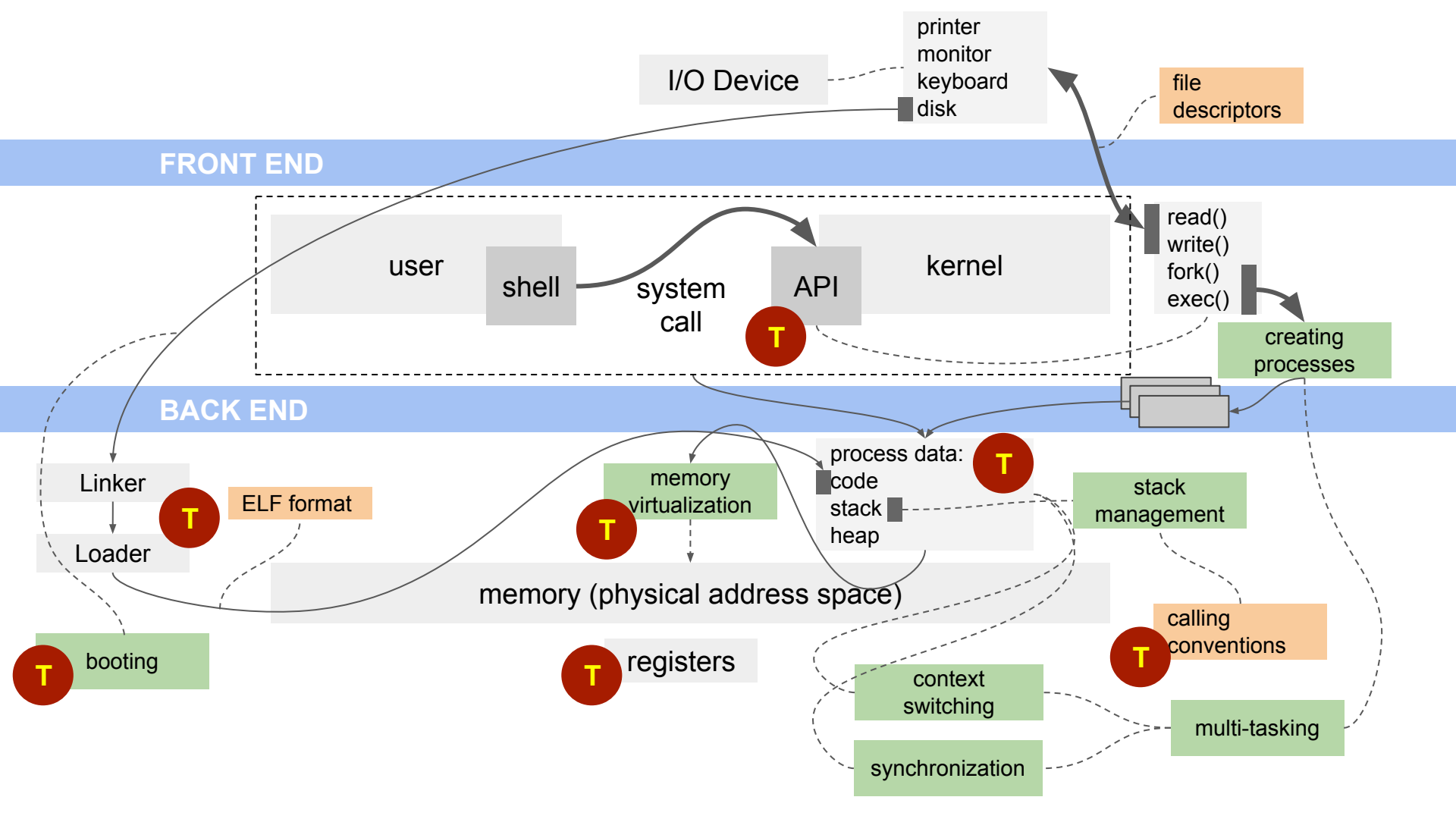

I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
BACK END
Linker
ELF format
Loader
memory
virtualization
process data:
code
stack
heap
stack
management
memory (physical address space)
booting
registers
calling
conventions
T

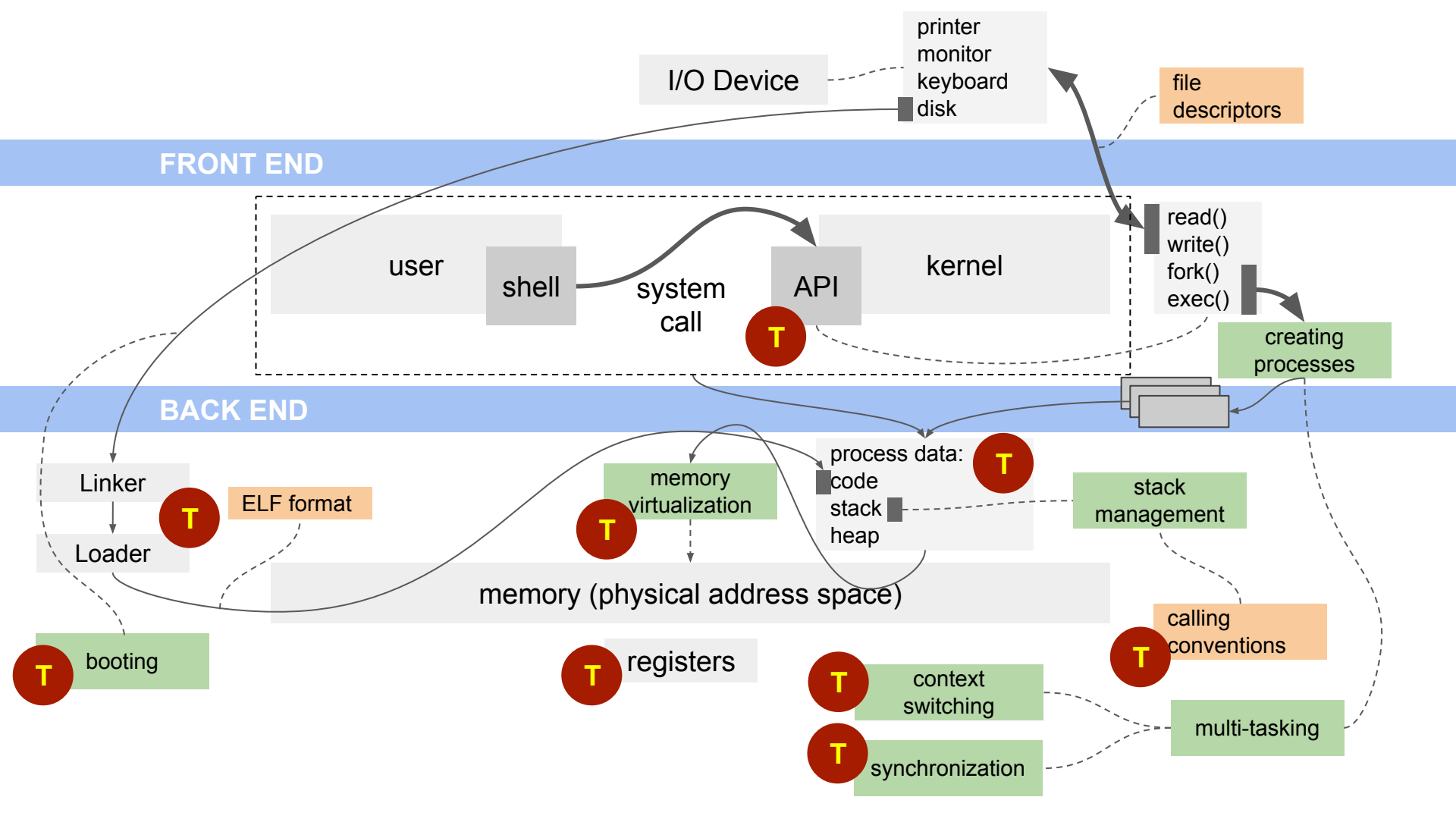
multi-tasking

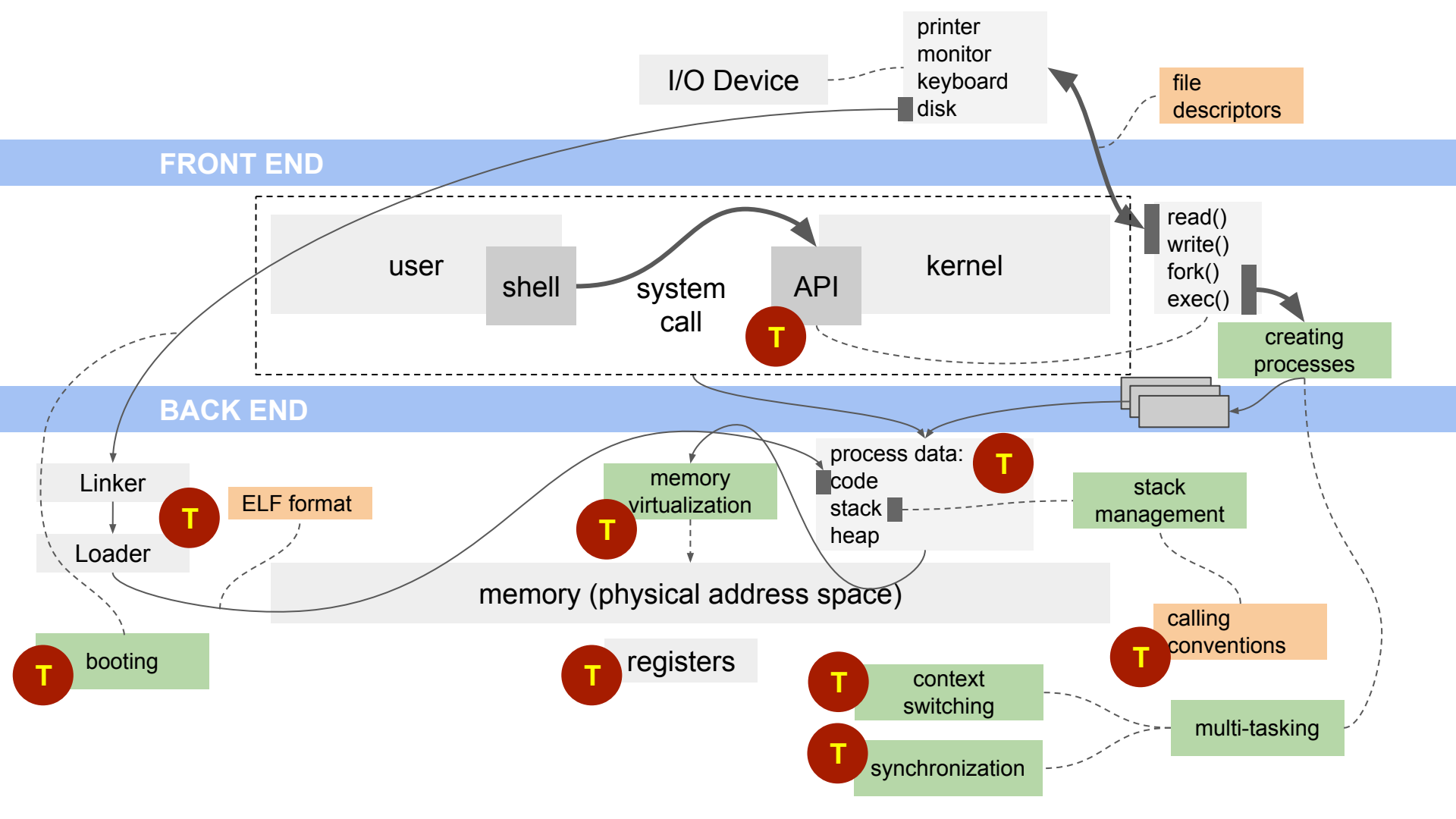

I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
T
BACK END
Linker
Loader
T
ELF format
memory
virtualization
T
process data:
code
stack
heap
T
stack
management
memory (physical address space)
T
booting
T
registers
calling
conventions
T
multi-tasking

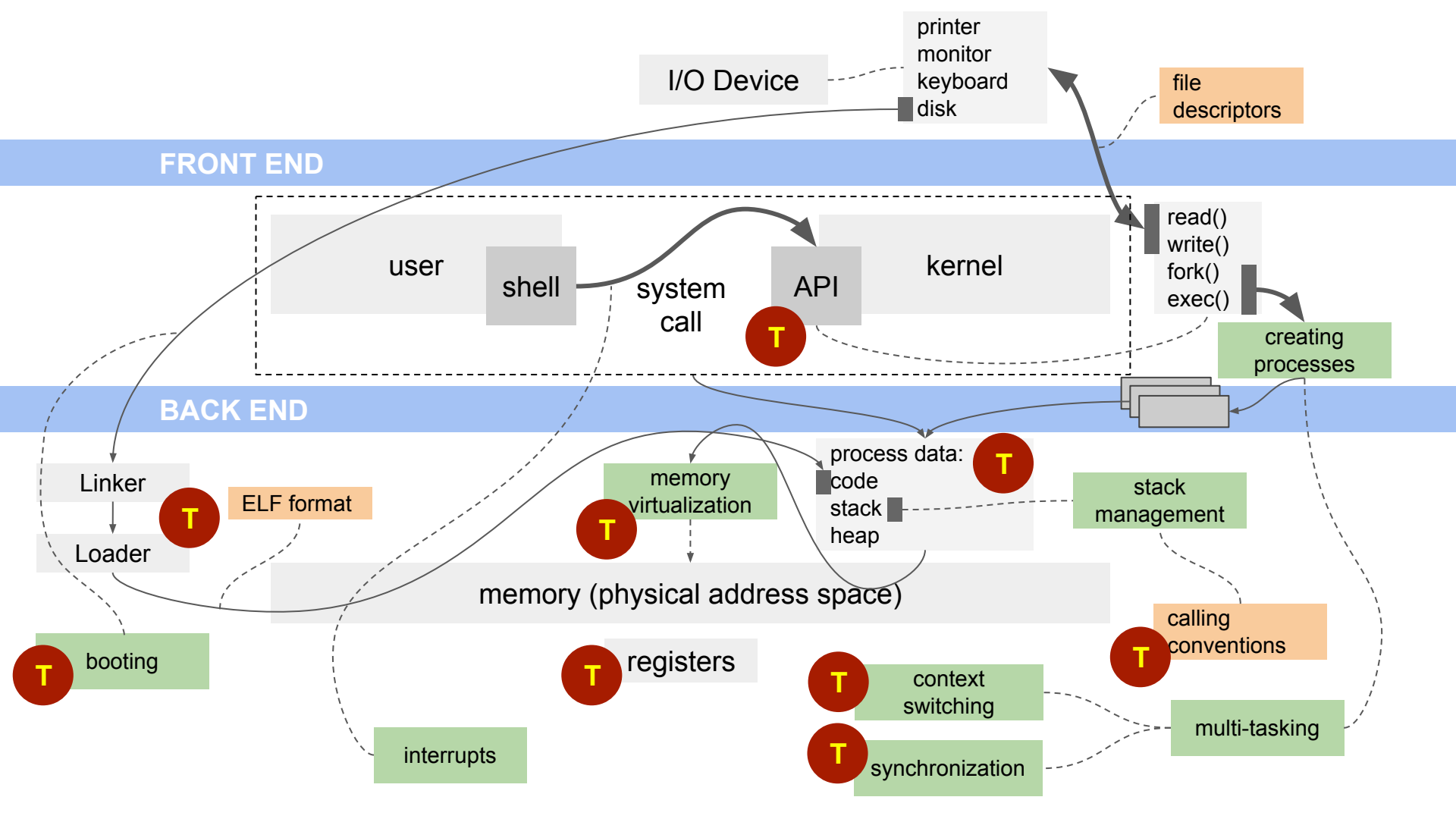

I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
T
BACK END
Linker
Loader
T
ELF format
memory
virtualization
T
process data:
code
stack
heap
T
stack
management
memory (physical address space)
T
booting
T
registers
T
calling
conventions
context
switching
multi-tasking

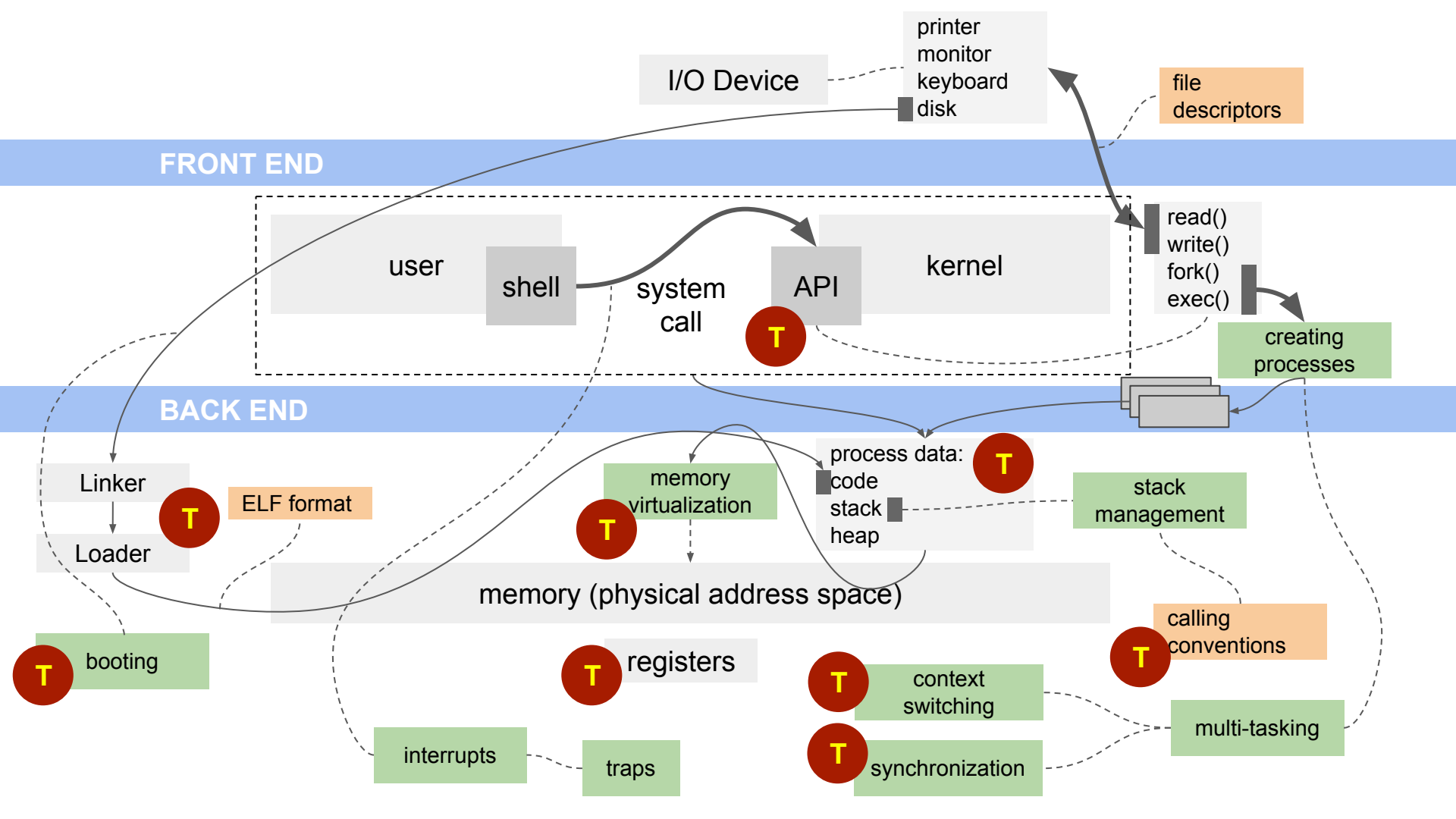

I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
BACK END
Linker
ELF format
Loader
memory
virtualization
process data:
code
stack
heap
stack
management
memory (physical address space)
booting
registers
calling
conventions
context
switching
multi-tasking
synchronization

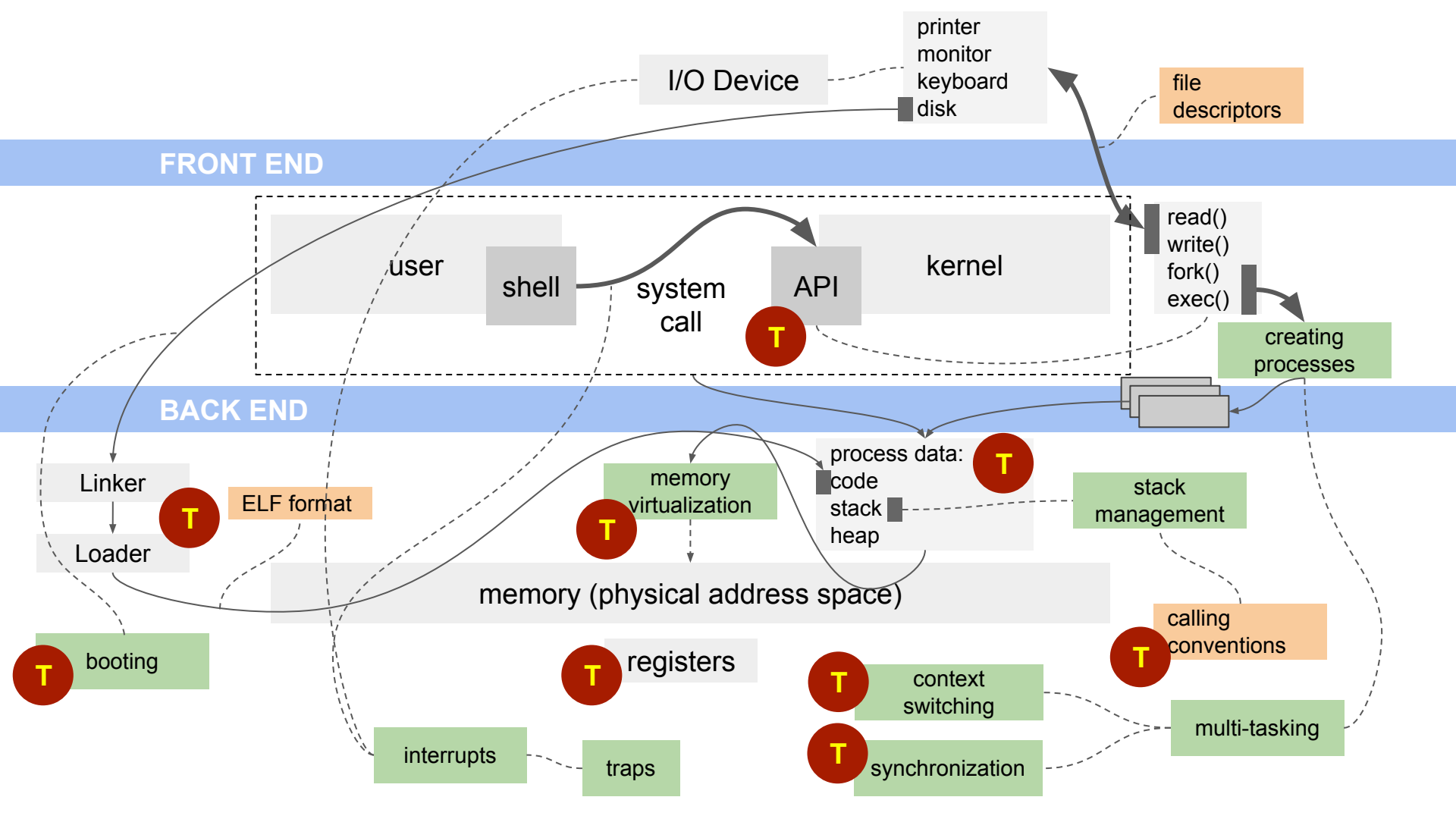

I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
T
BACK END
Linker
Loader
T
ELF format
memory
virtualization
T
process data:
code
stack
heap
T
stack
management
memory (physical address space)
T
booting
T
registers
calling
conventions
T
context
switching
multi-tasking
synchronization

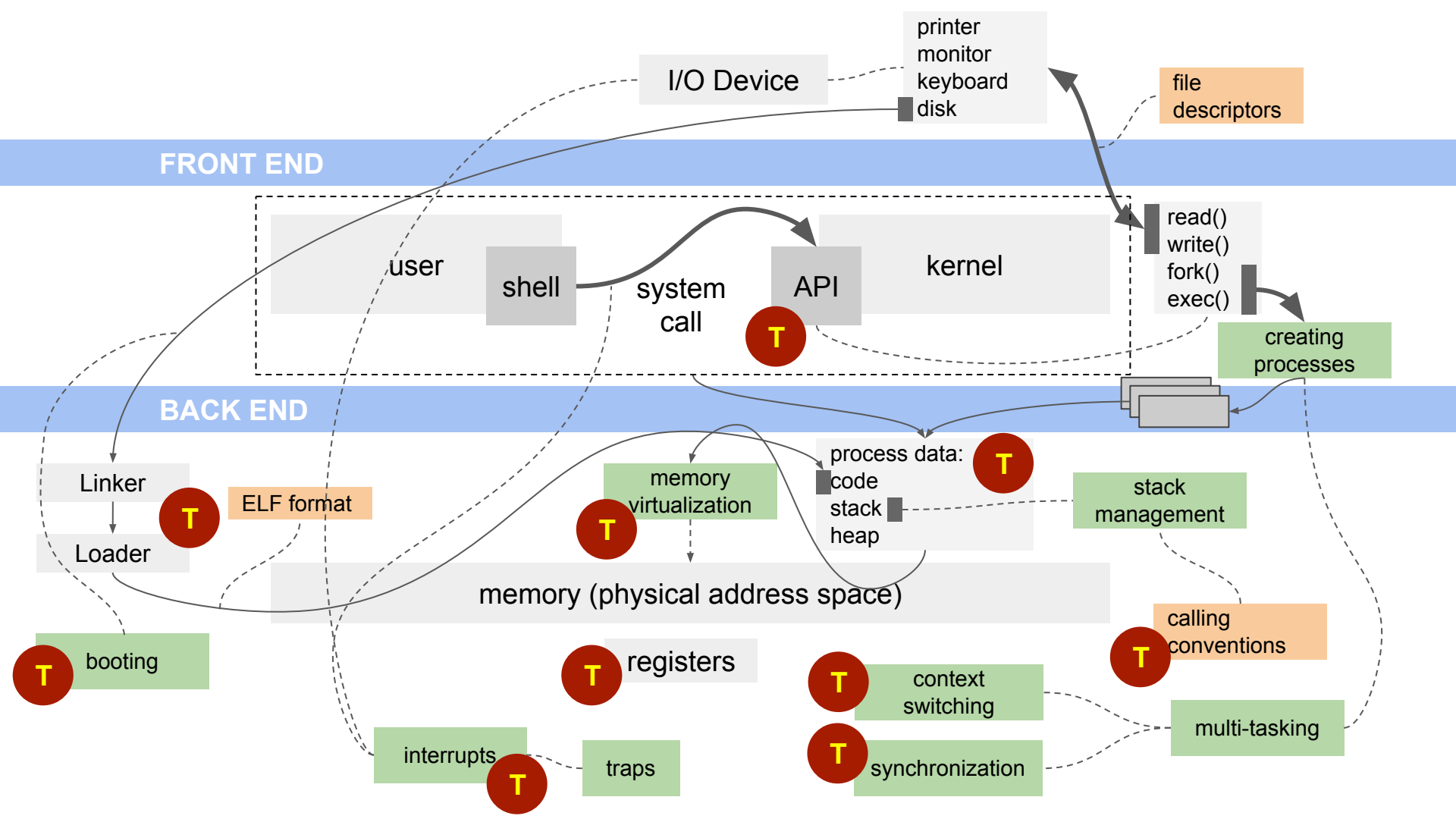

I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
T
BACK END
Linker
Loader
ELF format
memory
virtualization
process data:
code
stack
heap
stack
management
memory (physical address space)
booting
registers
calling
conventions
context
switching
multi-tasking
synchronization

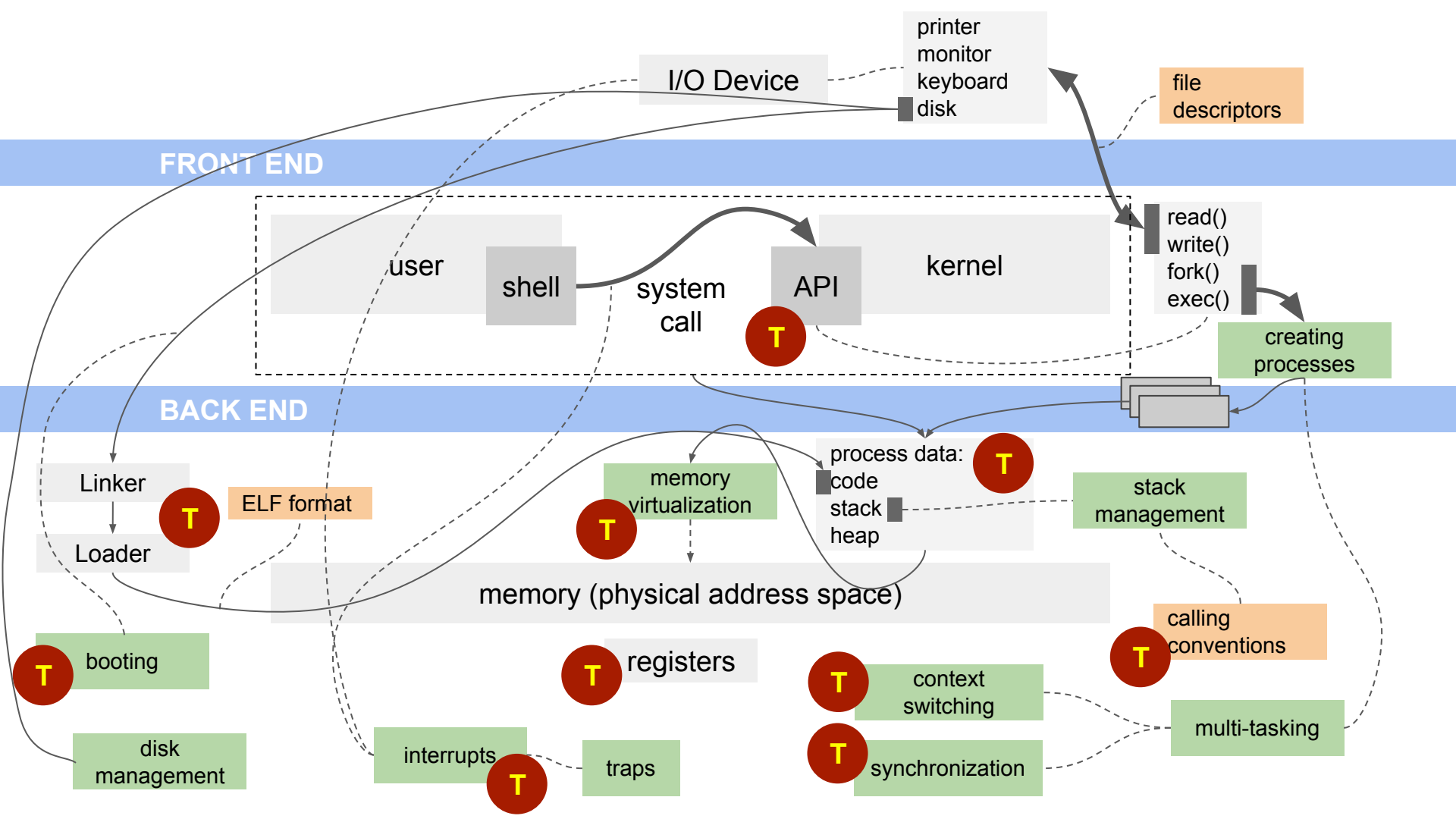

I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
T
BACK END
Linker
Loader
ELF format
memory
virtualization
process data:
code
stack
heap
stack
management
memory (physical address space)
booting
registers
calling
conventions
context
switching
multi-tasking
synchronization

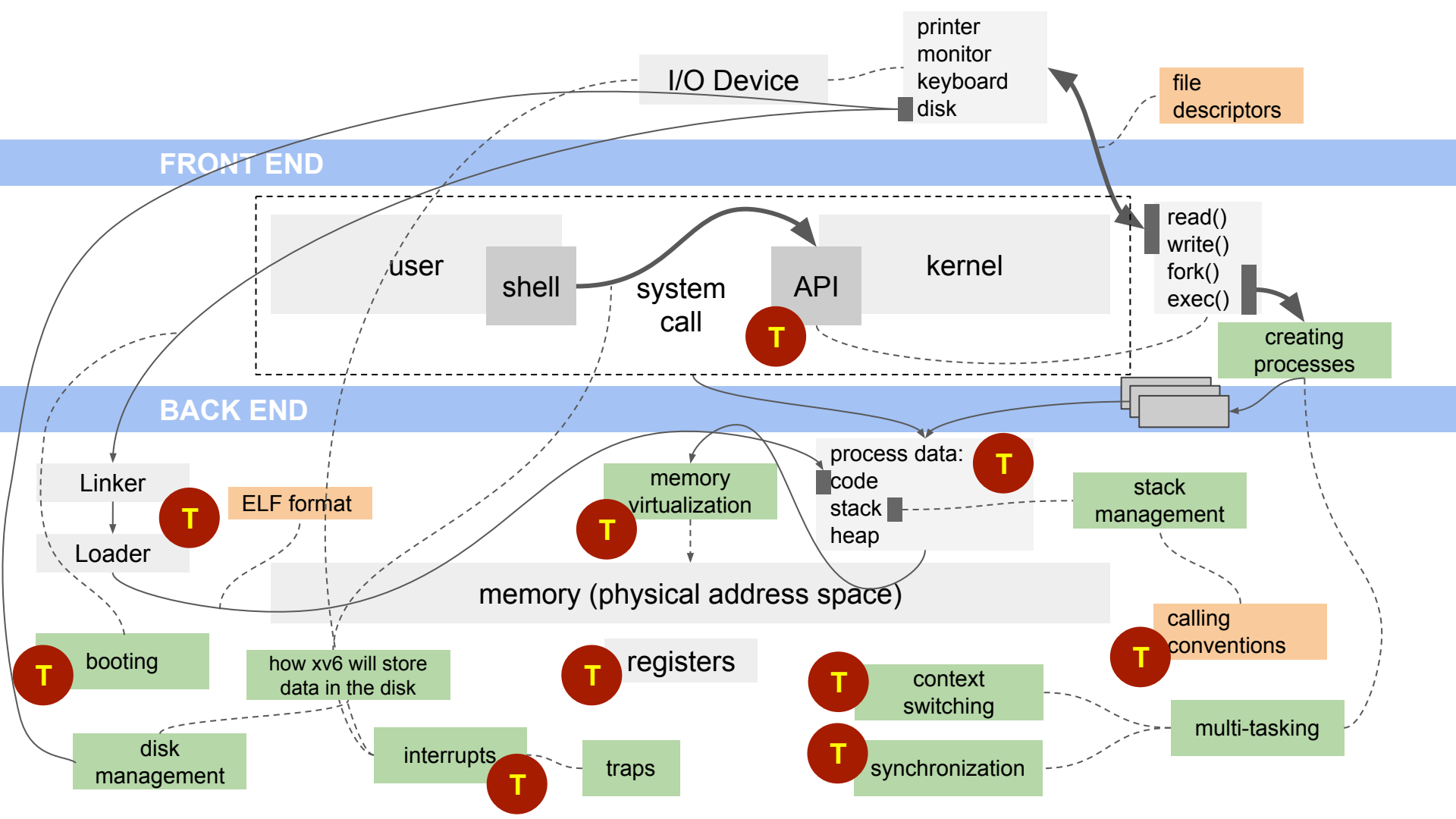

I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
BACK END
Linker
ELF format
Loader
memory
virtualization
process data:
code
stack
heap
stack
management
memory (physical address space)
calling
conventions
booting
registers
context
switching
multi-tasking
interrupts
synchronization
T

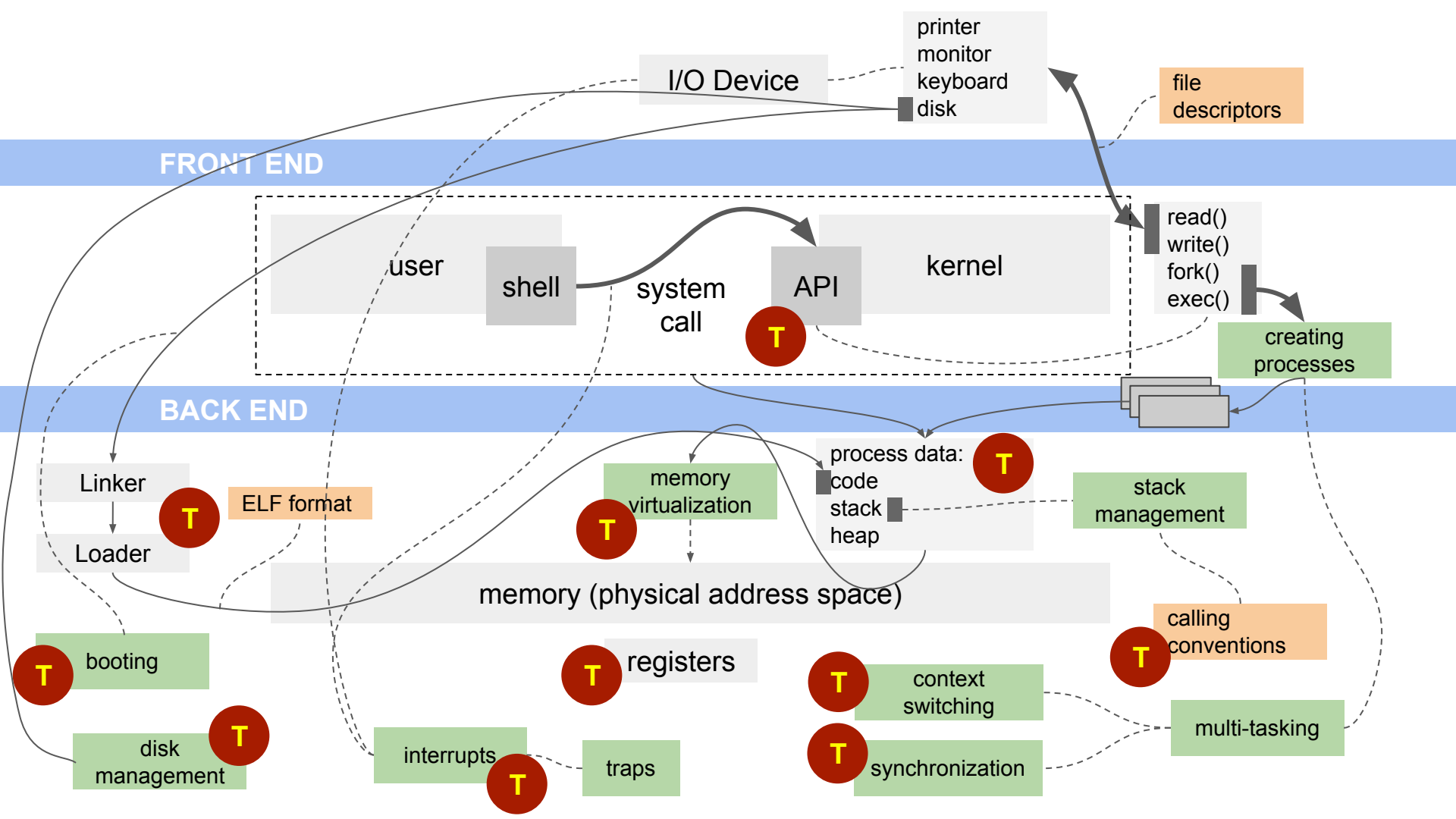

I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
BACK END
Linker
ELF format
Loader
memory
virtualization
process data:
code
stack
heap
stack
management
memory (physical address space)
booting
registers
calling
conventions
context
switching
multi-tasking
interrupts
traps
synchronization


I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
BACK END
Linker
ELF format
Loader
memory
virtualization
process data:
code
stack
heap
stack
management
memory (physical address space)
booting
registers
calling
conventions
context
switching
multi-tasking
interrupts
traps
synchronization

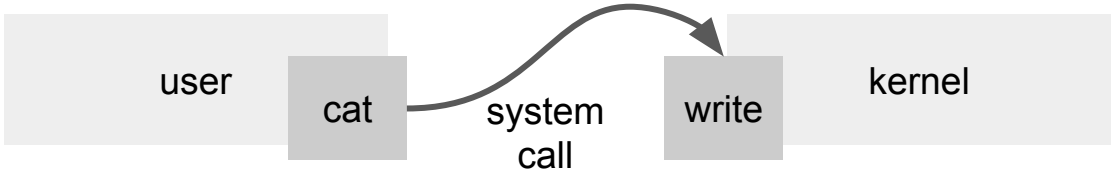

I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
BACK END
Linker
ELF format
Loader
memory
virtualization
process data:
code
stack
heap
stack
management
memory (physical address space)
booting
registers
calling
conventions
context
switching
multi-tasking
interrupts
traps
synchronization
T

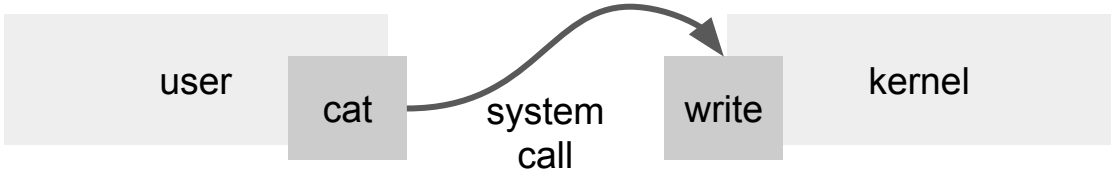

printer
monitor
keyboard
disk
I/O Device
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
BACK END
Linker
ELF format
Loader
memory
virtualization
process data:
code
stack
heap
stack
management
memory (physical address space)
calling
conventions
booting
registers
context
switching
multi-tasking
disk
management
interrupts
traps
synchronization
T


I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
BACK END
Linker
ELF format
Loader
memory
virtualization
process data:
code
stack
heap
stack
management
memory (physical address space)
calling
conventions
booting
how xv6 will store
data in the disk
registers
context
switching
multi-tasking
disk
management
interrupts
traps
synchronization
T


I/O Device
printer
monitor
keyboard
disk
file
descriptors
FRONT END
user
shell
system
call
API
kernel
read()
write()
fork()
exec()
creating
processes
BACK END
Linker
ELF format
Loader
memory
virtualization
process data:
code
stack
heap
stack
management
memory (physical address space)
calling
conventions
booting
registers
context
switching
multi-tasking
disk
management
interrupts
traps
synchronization
T

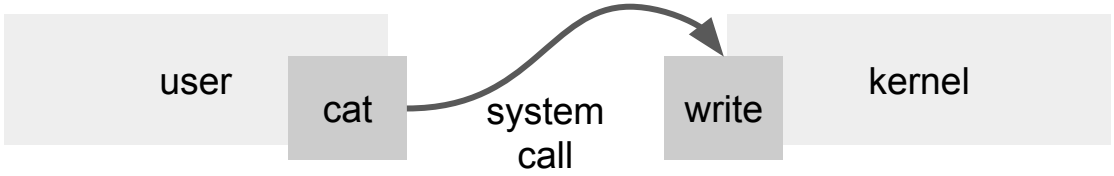
Now ... System Call Demo


user
cat
system
call
write
kernel



cat.c

```
if (write(1, buf, n) != n)
```



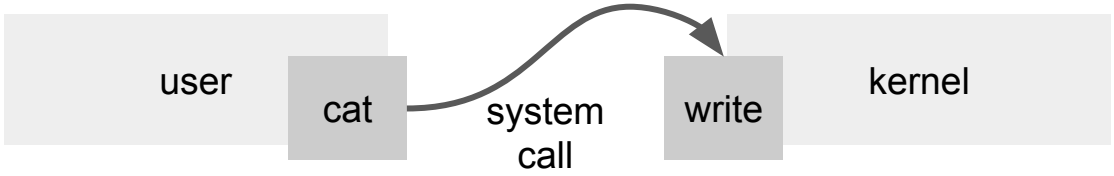
cat.c

```
if (write(1, buf, n) != n)
```

defined here

usys.S

```
SYSCALL(write)
```



cat.c

```
if (write(1, buf, n) != n)
```

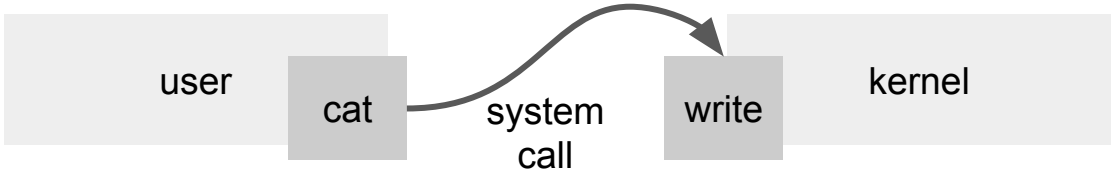
defined here

usys.S

```
SYSCALL(write)
```

expanded to

```
#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
```



cat.c

```
if (write(1, buf, n) != n)
```

defined here

usys.S

```
SYSCALL(write)
```

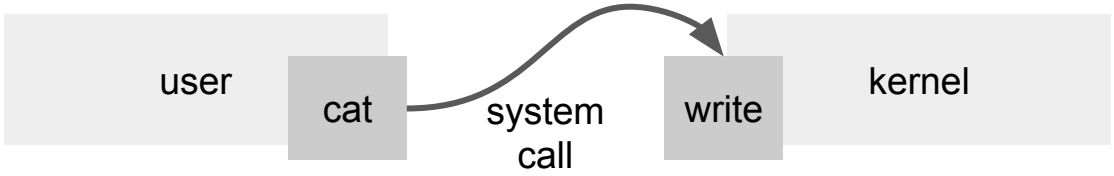
expanded to

```
#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
```

expansion provided here

syscall.h

```
#define SYS_write  16
```



cat.c

```
if (write(1, buf, n) != n)
```

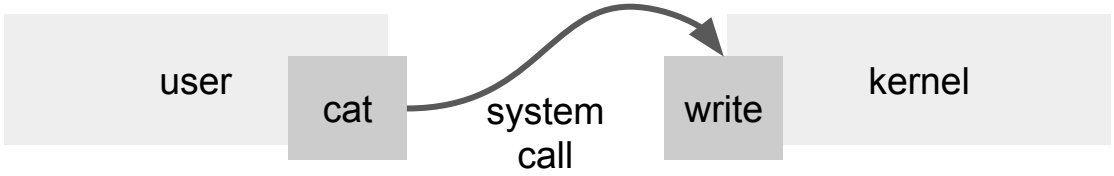
defined here

usys.S

```
SYSCALL(write)
```

expanded to

```
#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
```



cat.c

```
if (write(1, buf, n) != n)
```

defined here

usys.S

```
SYSCALL(write)
```

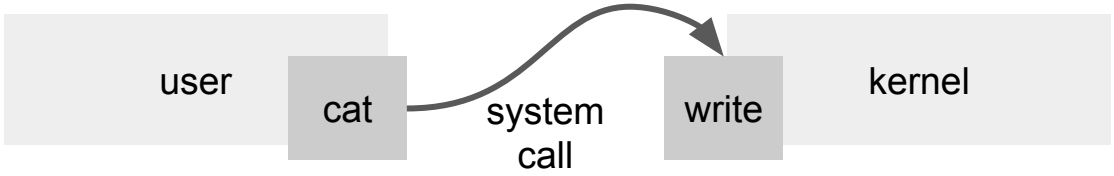
expanded to

```
#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
```

actual assembly
code generated here

usys.o

```
00000028 <write>:
  28:b8 10 00 00 00    mov    $0x10,%eax
  2d:cd 40             int    $0x40
  2f:c3                ret
```



cat.c

```
if (write(1, buf, n) != n)
```

defined here

usys.S

```
SYSCALL(write)
```

expanded to

```
#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
```

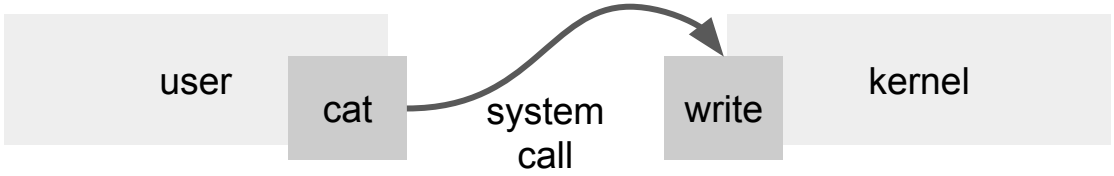
actual assembly code generated here

usys.o

```
00000028 <write>:
  28:b8 10 00 00 00    mov    $0x10,%eax
  2d:cd 40             int    $0x40
  2f:c3                ret
```

EAX

10



cat.c

```
if (write(1, buf, n) != n)
```

defined here

usys.S

```
SYSCALL(write)
```

expanded to

```
#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
```

actual assembly
code generated here

usys.o

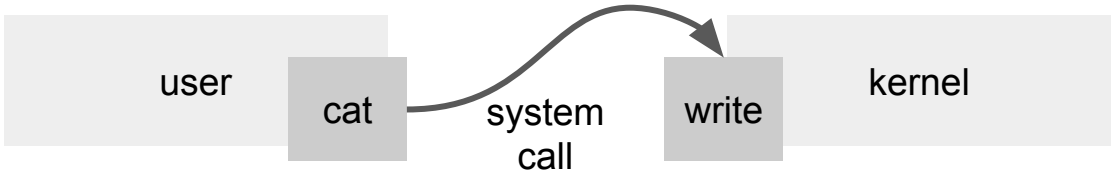
```
00000028 <write>:
  28:b8 10 00 00 00    mov    $0x10,%eax
  2d:cd 40             int    $0x40
  2f:c3                ret
```

EAX

10

vectors.S

```
jmp alltraps
```



cat.c

```
if (write(1, buf, n) != n)
```

defined here

usys.S

```
SYSCALL(write)
```

expanded to

```
#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
```

actual assembly
code generated here

usys.o

```
00000028 <write>:
  28:b8 10 00 00 00     mov     $0x10,%eax
  2d:cd 40              int     $0x40
  2f:c3                 ret
```

EAX

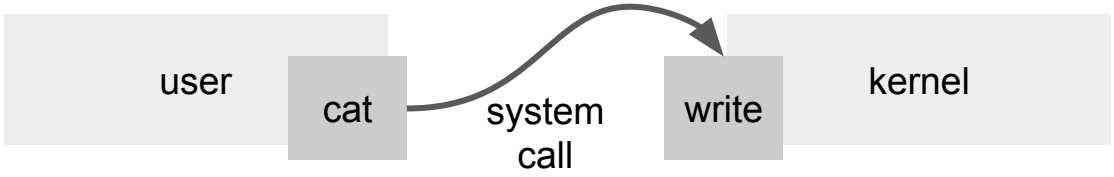
10

[trapasm.S](#)

defined in

vectors.S

```
jmp alltraps
```



cat.c

```
if (write(1, buf, n) != n)
```

defined here

usys.S

```
SYSCALL(write)
```

expanded to

```
#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
```

actual assembly code generated here

usys.o

```
00000028 <write>:
  28:b8 10 00 00 00     mov     $0x10,%eax
  2d:cd 40              int     $0x40
  2f:c3                 ret
```

EAX

10

trap.c

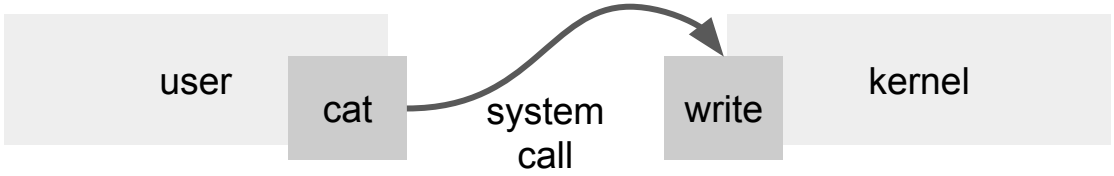
calls traps()

trapasm.S

defined in

vectors.S

```
jmp alltraps
```



cat.c

```
if (write(1, buf, n) != n)
```

defined here

usys.S

```
SYSCALL(write)
```

expanded to

```
#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
```

actual assembly code generated here

usys.o

```
00000028 <write>:
  28:b8 10 00 00 00     mov     $0x10,%eax
  2d:cd 40              int     $0x40
  2f:c3                 ret
```

EAX

10

syscall.c

calls syscall()

trap.c

calls traps()

trapasm.S

defined in

vectors.S

```
jmp alltraps
```

Thank you!