22

TENTAMEN (EXAMINATION)

Tentamensdatum/*Examination date:* 18-03-17
(åå-mm-dd/*yy-mm-dd*)

AID-nummer
AID number

Ifylles av student: 1690
Completed by student

Ifylles av vakt: 1690
Completed by supervisor

Kurskod/*Course code:* TDDB68 **Provkod/*Exam code:*** TEN1

Kursnamn/*Course title:* Processprogrammering och operativsystem

Institution/*Department:* IDA

Jag intygar att varken mobil eller något annat otillåtet hjälpmedel finns tillgängligt under tentamen.
I confirm that no mobile or other non-permitted aids are available during the examination. ☒

Inlämnat: antal lösblad 10 **tentamensformulär** ☐
Enclosed: number of sheets ***exam booklet***

Markera behandlade uppgifter med X/*Mark tasks attempted with an X*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
X här/*here*	X	X	X	X	X	X									
Erhållna poäng *Points obtained*	8	6	6	2	6	1.5									
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
X här/*here*															
Erhållna poäng *Points obtained*															

Anvisningar/*Instructions*

1. Skriv AID-nummer, datum, kurskod och provkod på varje blad som lämnas in/ *Write AID number, date, course code and exam code on every sheet that is handed in*
2. På varje papper får högst en uppgift lösas om inget annat anges/ *Maximum one task per sheet unless otherwise instructed*
3. Skriv endast på papprets ena sida om inget annat anges/ *Use only one side of each sheet unless otherwise instructed*
4. Numrera de papper som lämnas in/*Number every sheet that is handed in*
5. Använd inte röd penna/*Do not use a red pen/pencil*

Sen inlämning ☐
Late hand in

Klockslag
Time

Orsak
Reason

Σ Poäng/*Points:* 32,5 + 4 = 36,5 Betyg/*Grade:* 5

Examinator/*Examiner:* [signature]

LINKÖPINGS UNIVERSITET

AID: 1690 Date: 18-03-17 Sheetnumber: 1

Course code TDDB68 Exam code: TEN1

Multiple choice form for answering question 1. Please put X:s in the appropriate cells:

	A	B	C	D	
1 a)		X	X		1
1 b)	X		X		1
1 c)	X	X	X		0
1 d)	X	X			1
1 e)	X				0
1 f)		X	X	X	1
1 g)		X			1
1 h)		X			1
1 i)	X			X	1
1 j)	X		X		1

/8p

AID-nummer: AID-number: 1690	Datum: Date: 18-03-17	Blad nummer: Sheet number: 2
Kurskod: Course code: TDDB68	Provkod: Exam code: TEN1	

LINKÖPINGS UNIVERSITET

②

a) if we have 2 inserts concurrently:

i)

T1	T2
int i = h(u)	
struct htelem e* = ...	
	int i = h(u)
	struct htelem e* = ...
e->item = u	
e->next = T[i]	
	e->item = u
	e->next = T[i]
T[i] = e	
	T[i] = e

in this case, the insertion done by P1 is overwritten by P2.

ii)

T1	T2
int i = h(u)	
	int i = h(u)
struct htelem e* = ...	
	struct htelem e* = ...
e->item = u	
	e->item = u
e->next = T[i]	
T[i] = e	
	e->next = T[i]
	T[i] = e

1p

in this case P1 does the insertion first, then P2 does it.
=> two different interleavings don't give the same results and one of them is incorrect

b)

```
lock l;
void insert( ... )
{
  int i = h(u); struct htelem e* = ...;
  e->item = u
  acquire (l);
  e->next = T[i]
  T[i] = e
  release (l);
}
int lookup( ... )
{
  int i = h(u); struct htelem p*;
  acquire (l);
  for(p = T[i]; p != NULL; p = p->next)
    if (equal(u, p->item)) { release(l); return 1; }
  release(l);
  return 0;
}
```

(explanation on next page)

AID-nummer: *AID-number:* 1690	Datum: *Date:* 18-03-17
Kurskod: *Course code:* TDDB68	Provkod: *Exam code:* TEN1

Blad nummer: *Sheet number:* 3

② b)

The critical sections are during the insertion at the moment where a list is modified, and in the lookup when we go through a list. We just need to acquire a lock before the section and release it after. 2p

c) With only one lock, if a thread tries to insert an element with hash = 10, another thread won't be able to insert an element with hash = 11, even if they don't interfere with the same list. It basically means that our code will never be executed in parallel, which isn't optimal if many threads want to access it. 1p

d) we need a lock for every possible hash value, and when inserting or looking-up, we only acquire the lock corresponding to the hash value of our element, making concurrent inserts possible for different hashes. 1p

e) Use a reader-writer lock, that allow multiple readers (look-up) to run at the same time. However, we can only have one writer (insert) at the time, and it will prevent any read at the same time => we can read from the same hash-value list from multiple threads at the same time. 1p

/8p

③

a) Deadlock prevention: we make sure that the 4 Coffman conditions are never all met at the same time, we can for example prevent "Hold & wait" by forbidding processes to acquire and keep a ressource if they are waiting on another ressource.

(2)

Deadlock avoidance: every time a ressource is about to be allocated, we check if it will lead to a safe state from which we won't get into a deadlock. We can use Banker's algorithm to ensure that.

b) Allocated = $\begin{pmatrix} 0&0&0 \\ 2&1&2 \\ 1&0&0 \\ 0&4&0 \end{pmatrix}$ Max = $\begin{pmatrix} 0&1&0 \\ 3&3&3 \\ 1&3&2 \\ 1&5&0 \end{pmatrix}$ => need = $\begin{pmatrix} 0&1&0 \\ 1&2&1 \\ 0&3&2 \\ 1&1&0 \end{pmatrix}$

Available = (0 3 2) Request P2 = (0 2 0)

Request ≤ need[2] ? yes | Request ≤ Available? yes

=> we assume we allocate, we then have:

Allocated = $\begin{pmatrix} 0&0&0 \\ 2&3&2 \\ 1&0&0 \\ 0&4&0 \end{pmatrix}$ need = $\begin{pmatrix} 0&1&0 \\ 1&0&1 \\ 0&3&2 \\ 1&1&0 \end{pmatrix}$ Available = (0 1 2)

Let's check if this is a safe state: Work = (0 1 2), Finish = (FFFF)

P1 can finish since need[1] ≤ work => it releases it's allocated ressources

=> work = (0 1 2), Finish = (TFFF)

P2 can't finish since need[2] = (1 0 1) ≰ work = (0 1 2)

P3 can't finish since need[3] = (0 3 2) ≰ work = (0 1 2)

P4 can't finish since need[4] = (1 1 0) ≰ work = (0 1 2)

=> Finish = (TFFF) so this isn't a safe state => we don't grant the request to P2.

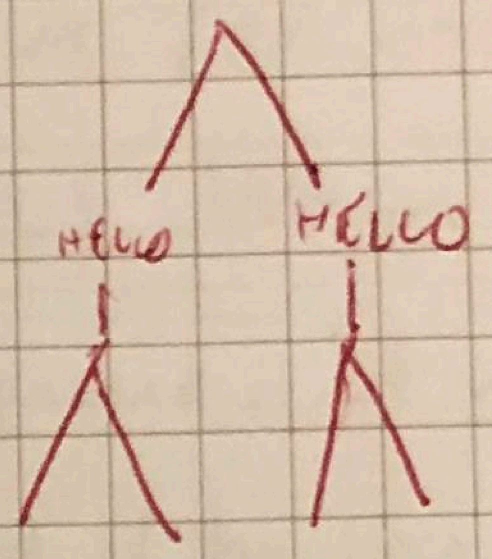
(4)

AID-nummer: / *AID-number:* 1690	Datum: / *Date:* 18-03-17
Kurskod: / *Course code:* TDDB68	Provkod: / *Exam code:* TEN1

Blad nummer: / *Sheet number:* 5

(4)

a)
```
int main () {
    process 1 = fork();
    exec (process1, print("Hello world"))
    process 2 = fork()
    exec (process2, print ("Hello world"))
    wait(process 1);
    wait(process 2);
    print("Goodbye");
    return 0;
}
```



0.5

The fork() command will create a new child process and copy the content from the parent. Calling exec(p, function) will replace the content of a process by the given function and run it. (here we replace the child's code by the content of the print function). We do this twice, once for each child.
We then wait for both children process to finish using wait().

b) Message passing will have less overhead. With shared memory, there will be a lot of waiting due to the locks used for synchronization. When both processes want to write, one will have to wait. With message passing, they can just send their messages whenever they want, they never need to wait on the other process to release the lock.

AID-nummer: / AID-number: 1690	Datum: / Date: 18-03-17	Blad nummer: / Sheet number: 6
Kurskod: / Course code: TDDB68	Provkod: / Exam code: TEN1	

(4)

c) This is a multilevel queue that gives higher priority to short processes. We can have for instance three queues: High(H), Medium(M) and Low(L). When a new process arrives, it is placed in the H queue. It will run for a duration of a time quantum q, and if by the end it isn't finished, it is moved to the M queue. In the M queue, the same thing will be applied. The time allocated for each queue can be proportional to its importance (e.g 60% for H, 25% for M and 15% for L). It is often used in general purpose OS because most processes that matter for the speed and responsiveness of the OS are short, wheras we don't care if already slow processes are executed a bit later (copying a file, downloading a video, ...) We don't want short processes to be stuck behind long processes.

AGING?

STARVATION?

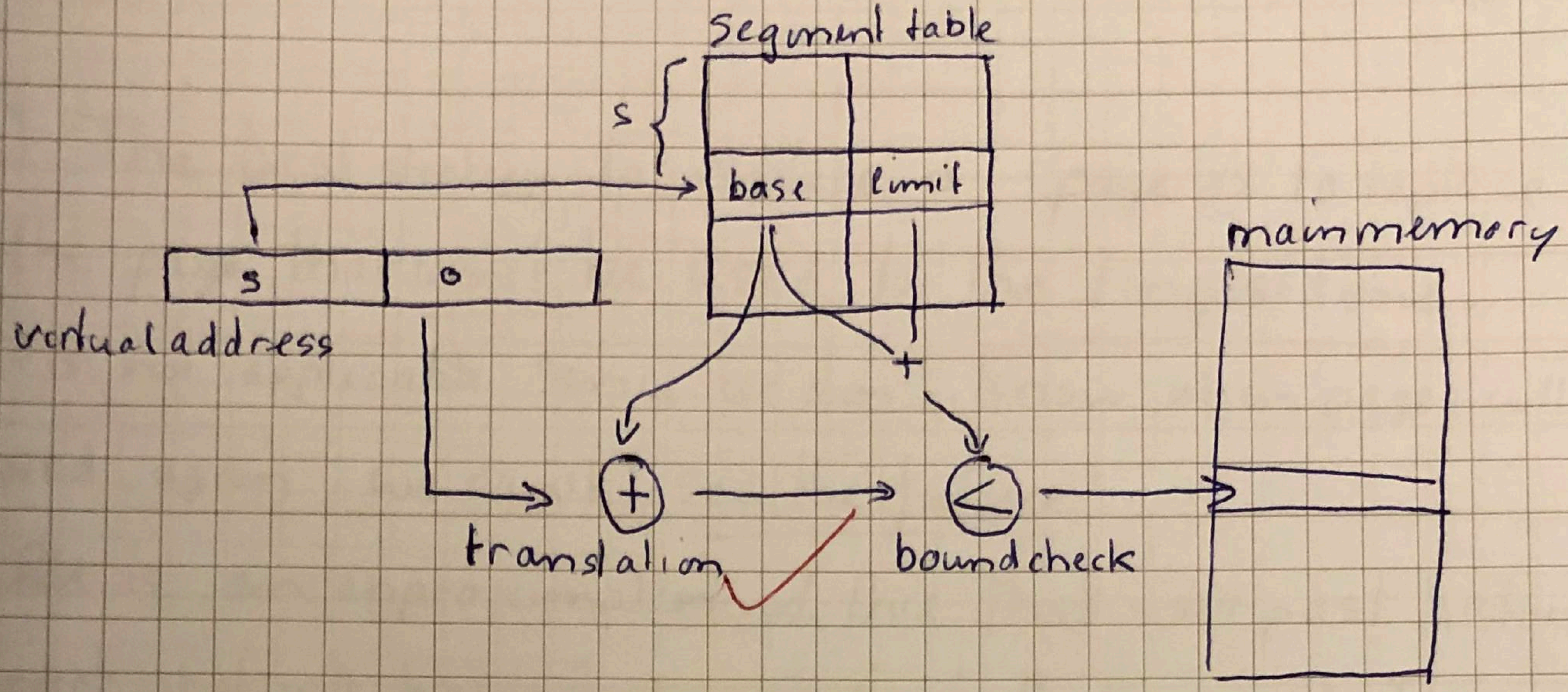
1.5

ÖPINGS UNIVERSITET

AID-nummer: / AID-number: 1690	Datum: / Date: 18-03-17	Blad nummer: / Sheet number: 7
Kurskod: / Course code: TDDB68	Provkod: / Exam code: TEN1	

⑤

a)



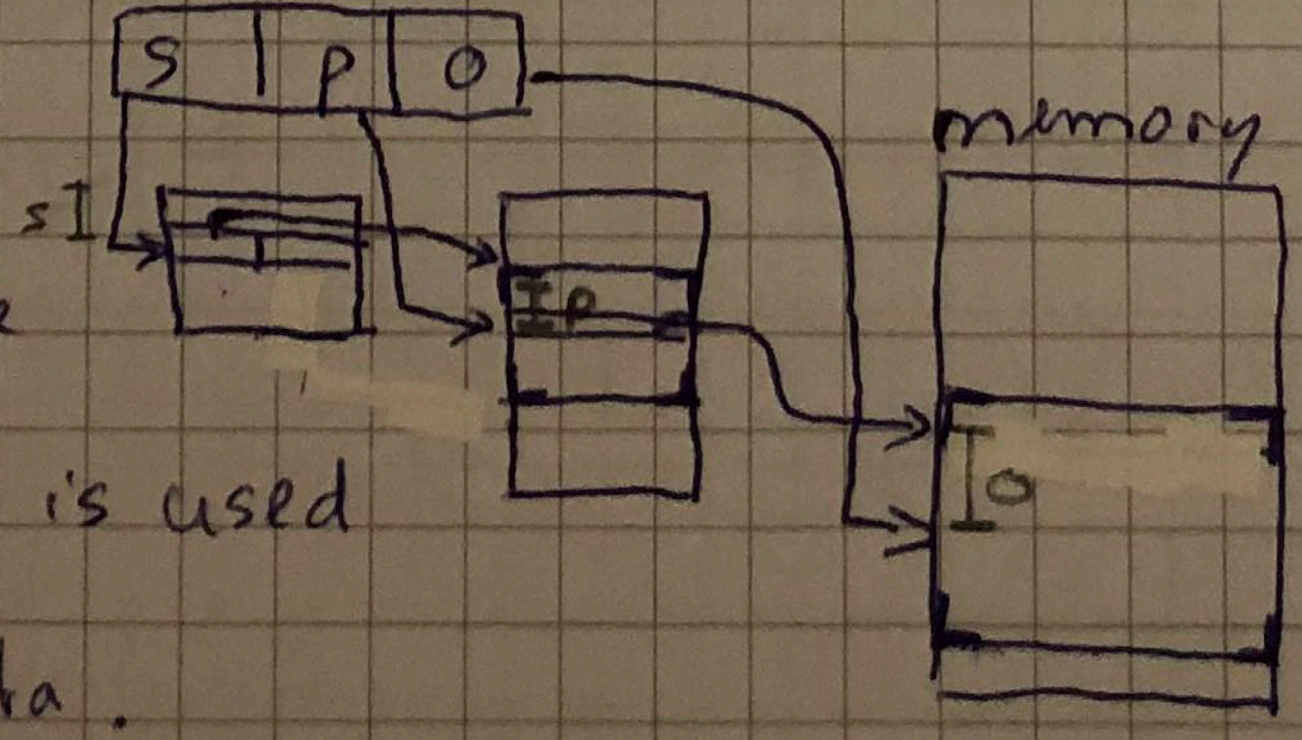
From the virtual address, the first bits represent the segment number. It is used to find a base and a limit in the segment table. The base indicates the position of the segment in memory, so it's added to the offset. We then check that the address doesn't exceed the limit. All this (address translation and bound checking for security) is done in hardware

2

b) if we have virtual addresses like this: | s | p | o | :

s, the segment number, will be used to find the address of a page table, from where we can use p (page #) to find the frame in the memory.

Once we have the page in memory, the offset is used to find the correct data.



1

AID-nummer: AID-number: 1690	Datum: Date: 18-03-17
Kurskod: Course code: TDDB68	Provkod: Exam code: TEN1

Blad nummer: Sheet number: 8

(5)

c) The best strategy to replace a page is to replace the page that won't be used for the longest time.
It is not applicable since we don't know when pages will be used again, we can't "see the future".
LRU is an approximation of this that uses past history to predict what will happen, by assuming that pages that haven't been used for a long time won't be used for a long time. Whereas this optimal technique would actually look at the future to get an exact knowledge of which process won't be used for the longest time

(3)

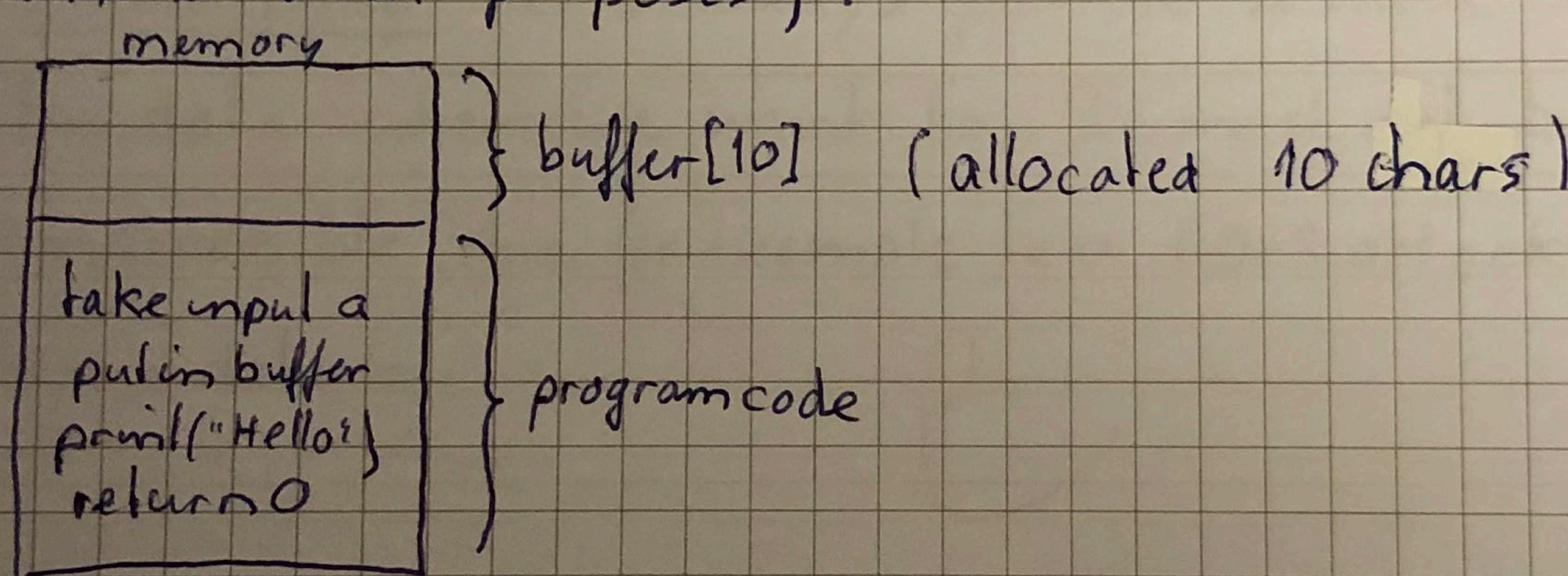
AID-nummer: / AID-number: 1690	Datum: / Date: 18-03-17
Kurskod: / Course code: TODB68	Provkod: / Exam code: TEN1

Blad nummer / Sheet number: 9

LINKÖPINGS UNIVERSITET

⑥

a) A buffer-overflow attack [can] happens when a program stores some data in a buffer where the size of the buffer is smaller than the size of the data. In the memory, it means that this overflow of data will store itself in the memory allocated to other variables, aswell as in extreme cases to the return address and even the code itself. Let's imagine we have the following data in memory (not like this in real systems, just for demonstration purposes):



The user runs the program and give the input:
"abcdefghij int main{ hack_system(); return 0; }"
The first 10 characters will be put in the buffer, but the rest will overflow in the memory of the code itself and replace it, and then be executed. The hijacker can put anything instead of hack_system(); and it will be run with the same authorization as the original program.

1.5

This can only happen when we do not check the size of the buffer and the data to make sure enough memory is allocated.

AID-nummer: *AID-number:* 1690	Datum: *Date:* 18-03-17	Blad nummer: *Sheet number:* 10
Kurskod: *Course code:* TDDB68	Provkod: *Exam code:* TEN1	

LINKÖPINGS UNIVERSITET

⑥

b) Confidentiality: confidential data can only be seen by authorized people. People that don't have the right to see it won't be able to.

Integrity: Some data can only be changed by authorized people. They might be able to read it, but not to change it. If they manage to change it, other people will be able to see that an unauthorized change has been made.

Availability: It is always possible to have access to the data when we want to. It must not be unaccessible due for exemple to a DDoS attack, or some data loss.