Norwegian University of Science and Technology

Operating Systems

Theoretical Exercise 4: Solutions and Discussion and some tips for PE4…

Michael Engel

4.1 Buddy algorithm

A memory management system is allocated using the Buddy algorithm for a memory with a total size of 512 kB and a minimum block size of 64 kB

Enter the resulting memory layout at t = 2 in the table. If an allocation or release cannot be performed, indicate this in the respective table.

a. Scenario 1:

The 64kB block b is released, but no combination to a larger block with a size of a power of 2 is possible.

t	Operation	Block	Size	64 kB	64 kB	64 kB	64 kB	64 kB	64 kB	64 kB	64 kB
1	Initial →			128 kB		a	b	256 kB			
2	R	b	—	128 kB		a	64 kB	256 kB			

b. Scenario 2:

t	Operation	Block	Size	64 kB	64 kB	64 kB	64 kB	64 kB	64 kB	64 kB	64 kB
1	Initial →			512 kB							
2	A	x	121 kB	x		128 kB		256 kB			

The 121 kB allocation for x requires a block of size 128 kB.
The remaining blocks must have a size of a power of 2, so we have two free blocks of 128 kB and 256 kB size, respectively.

4.1 Buddy algorithm

A memory management system is allocated using the Buddy algorithm for a memory with a total size of 512 kB and a minimum block size of 64 kB

Enter the resulting memory layout at t = 2 in the table. If an allocation or release cannot be performed, indicate this in the respective table.

c. Scenario 3: The 64kB block y is released, so a contiguous area of 256 kB can be combined

t	Operation	Block	Size	64 kB	64 kB	64 kB	64 kB	64 kB	64 kB	64 kB	64 kB
1	Initial →			128 kB		64 kB	y	a			
2	R	y	—	256 kB				a			

d. Scenario 4:

t	Operation	Block	Size	64 kB	64 kB	64 kB	64 kB	64 kB	64 kB	64 kB	64 kB
1	Initial →			128 kB		64 kB	a	b	64 kB	128 kB	
2	A	z	180 kB								

No allocation possible, since the 180 kB allocation would require a free block of at least 256 kB! The first two blocks (128 kB + 64 kB) don't suffice, since this would create an allocated part with a size that is not a power of 2.

4.2 First fit algorithm

Use the first fit strategy to implement the following sequence of memory requests. Note your results by completing the following table. Each field of the table stands for 1 MB of memory and there are 32 MB of memory available altogether:

a. release A, (already shown)
b. allocate 4 MB for F, (already shown)
c. allocate 2 MB for A,
d. release B,
e. release E,
f. allocate 7 MB for E,
g. release E,
h. allocate 4 MB for E

Initial layout	A	A	A	B	B	B			C	C	C	C	C	C							D	D	D	D	D	E	E	E	E			
Release A				B	B	B			C	C	C	C	C	C							D	D	D	D	D	E	E	E	E			
Alloc. F (4 MB)				B	B	B			C	C	C	C	C	C	F	F	F	F			D	D	D	D	D	E	E	E	E			
Alloc. A (2 MB)	A	A		B	B	B			C	C	C	C	C	C	F	F	F	F			D	D	D	D	D	E	E	E	E			
Release B	A	A		–	–	–			C	C	C	C	C	C	F	F	F	F			D	D	D	D	D	E	E	E	E			
Release E	A	A							C	C	C	C	C	C	F	F	F	F			D	D	D	D	D	–	–	–	–			
Alloc. E (7 MB)	A	A							C	C	C	C	C	C	F	F	F	F			D	D	D	D	D	E	E	E	E	E	E	E
Release E	A	A							C	C	C	C	C	C	F	F	F	F			D	D	D	D	D	–	–	–	–	–	–	–
Alloc. E (4 MB)	A	A	E	E	E	E			C	C	C	C	C	C	F	F	F	F			D	D	D	D	D							

4.3 Page replacement

Complete the given table using the first-in first-out (FIFO) approach. The age of each page frame is given as support information, you don't have to fill it in.

Allocation sequence →	1	2	3	4	5	6	1	2	3	2
Page frame	**1**	1	1	1	5	5	5	5	3	3
Page frame 2		**2**	2	2	2	6	6	6	6	6
Page frame 3			**3**	3	3	3	1	1	1	1
Page frame 4				**4**	4	4	4	2	2	2
Age of page frame 1 (optional)	0	1	2	3	0	1	2	3	0	1
Age of page frame 2 (optional)	>	0	1	2	3	0	1	2	3	4
Age of page frame 3 (optional)	>	>	0	1	2	3	0	1	2	3
Age of page frame 4 (optional)	>	>	>	0	1	2	3	0	1	2

No replacement here, page 2 is already in memory!

4.3 Page replacement

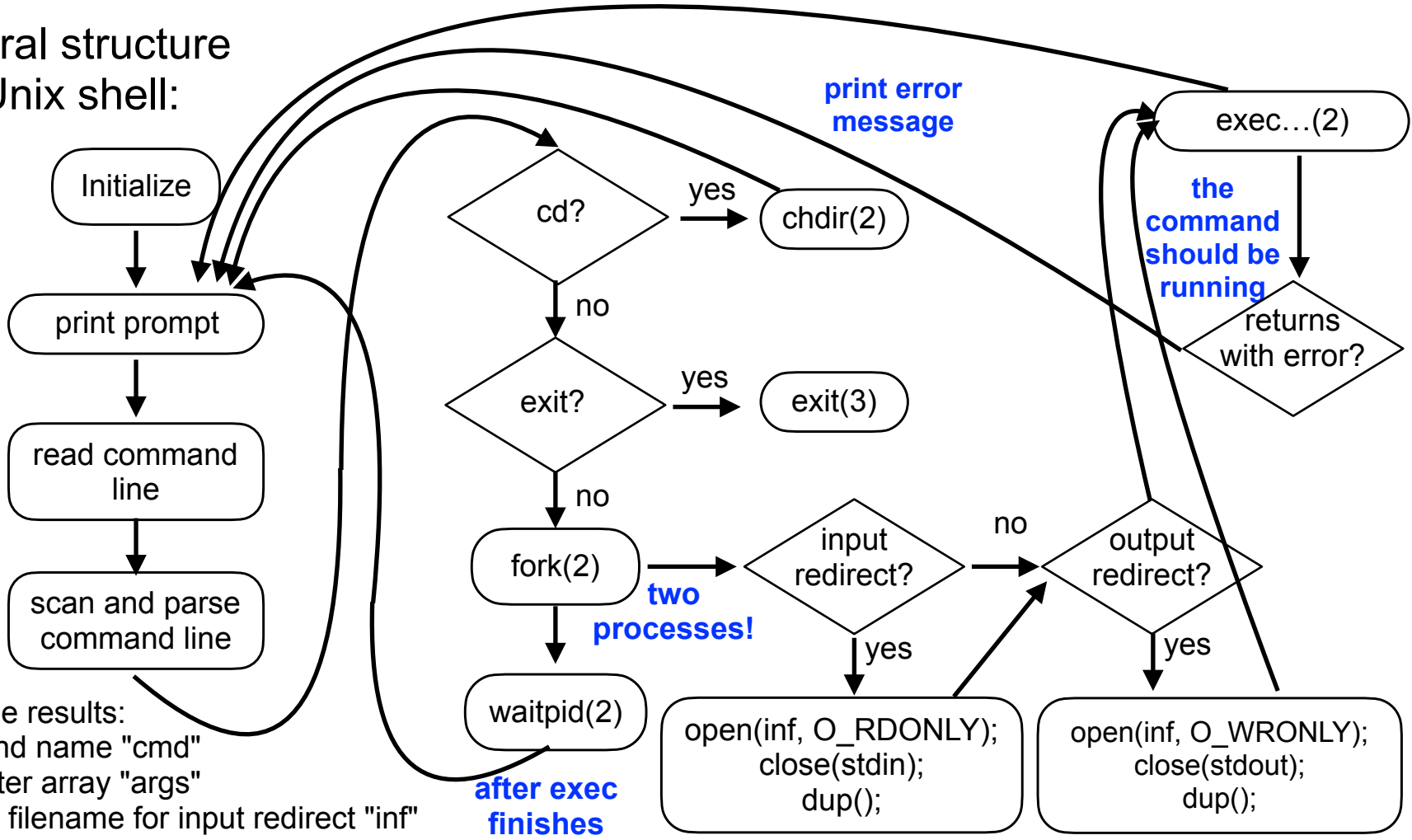
Complete the given table using the first-in first-out (FIFO) approach. The age of each page frame is given as support information, you don't have to fill it in.

Allocation sequence →	1	2	3	4	5	6	1	2	3	2
Page frame	**1**	1	1	1	5	5	5	5	3	3
Page frame 2		**2**	2	2	2	6	6	6	6	6
Page frame 3			**3**	3	3	3	1	1	1	1
Page frame 4				**4**	4	4	4	2	2	2
Age of page frame 1 (optional)	0	1	2	3	0	1	2	3	0	1
Age of page frame 2 (optional)	>	0	1	2	3	0	1	2	3	4
Age of page frame 3 (optional)	>	>	0	1	2	3	0	1	2	3
Age of page frame 4 (optional)	>	>	>	0	1	2	3	0	1	0

No replacement here, page 2 is already in memory!

PE4: Unix shell

General structure of a Unix shell:

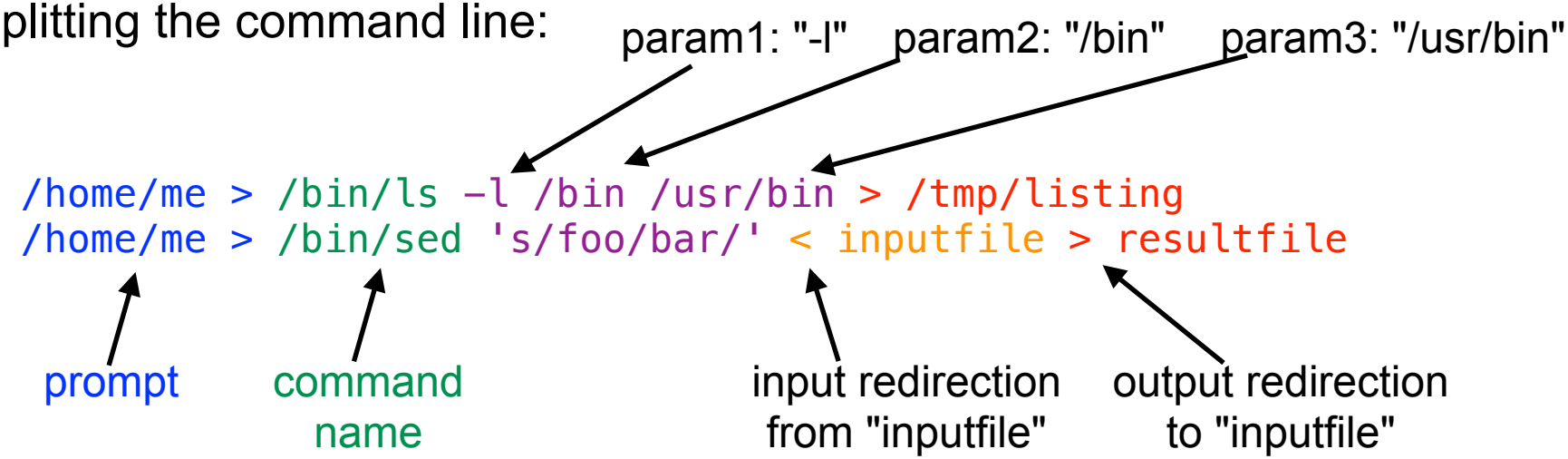


scan/parse results:

- command name "cmd"
- parameter array "args"
- optional filename for input redirect "inf"
- optional filename for output redirect "outf"

PE4: Unix shell parsing

Splitting the command line:



- Today, this would be called a REPL – "Read-Evaluate-Print-Loop"
- Prompt: what the shell prints
- Command name: command to execute (internal or external)
- Optional: zero or more parameters
- Optional: input and output redirect (in arbitrary order)

PE4: Unix shell parsing

Parsing by hand is lots of work and error-prone…

- Alternative: one of the strtok(3) libc functions
- From the strtok manpage on strtok_r(3):

```
char *
strtok_r(char *restrict str, const char *restrict sep, char **restrict lasts);

char line[80];
char *sep = "\\/:;=-";
char *word, *phrase, *brkt, *brkb;

strcpy(test, "This;is.a:test:of=the/string\\tokenizer-function.");

for (word = strtok_r(test, sep, &brkt); // strtok_r has an internal state machine
     word;
     word = strtok_r(NULL, sep, &brkt)) // it stores current pos in string in brkt
{
    printf("So far we're at %s:%s\n", word); // word contains ptr to current part
}
```

PE4: Unix shell parsing

Parsing by hand is lots of work and error-prone…

- Alternative: strsep(3)

```
char *
strsep(char **stringp, const char *delim);

First example:
char *token, *string, *tofree;

tofree = string = strdup("abc,def,ghi");
assert(string != NULL);

while ((token = strsep(&string, ",")) != NULL)
    printf("%s\n", token);

free(tofree);

Second example:
char **ap, *argv[10], *inputstring;

for (ap = argv; (*ap = strsep(&inputstring, " \t")) != NULL;)
    if (**ap != '\0')
        if (++ap >= &argv[10])
            break;
```

PE4: Unix shell I/O redirection

Redirecting I/O in Unix works uses the dup(2) or dup2(2) syscall:

```
int
dup(int fildes);

int
dup2(int fildes, int fildes2);
```

- dup copies the file descriptor passed as parameter to the first ***unused*** file descriptor

- to redirect I/O:
  - open the file you want to redirect to/from ➛ file descriptor, e.g. refd
  - then either close the fd you want to redirect (e.g. stdout = 1) and
    - and call dup with refd as parameter
  - or call dup2 with the fd you want to redirect and refd as parameters

PE4: Unix shell exec calls

There are several different exec functions in libc:

```
int
execl(const char *path, const char *arg0, ... /*, (char *)0 */);

int
execle(const char *path, const char *arg0, ... /*, (char *)0, char *const envp[] */);

int
execlp(const char *file, const char *arg0, ... /*, (char *)0 */);

int
execv(const char *path, char *const argv[]);

int
execvp(const char *file, char *const argv[]);

int
execvP(const char *file, const char *search_path, char *const argv[]);
```

- Depending on your representation of the parameters you parse, some might be more appropriate than others…