PDS/SDP OS internals 4/9/2023 – STANDARD version (with proposed solution)

PDS: L'esame è comune ai corsi PDS e SDP (in inglese). Si è predisposta un'unica proposta di soluzione in inglese.
SDP ON/OFF: As the ON/OFF exam is a subset of the standard one, the ON/OFF solution can be easily extracted from the standard one.

1 Consider the following program fragment*:*

```
#define N 32

data_t M[2*N][N];
int i,j,k;
...
for (k=0; k<4; k++) {
  for (i=0; i<2*N; i++) {
    for (j=k; j<N; j+=4) {
      M[i][j].diag = i-j;
    }
  }
}
```

The machine code generated from the program is executed on a system with memory management based on demand paging, 2KB pages, page replacement driven by a Working set (exact version) policy, with delta=10. Let's assume that:

- `sizeof(data_t) == 1024` is true
- `data_t` has a `diag` field of type `int` (size 32 bit)
- the code segment (machine instructions) has size less than one page
- `M` is allocated starting at logical address 0x5820C800
- the `M` matrix is allocated following the "row major" strategy, that is by rows (first row, followed by second row, …).

Answer the following questions:

A) How many pages (and frames) are needed to store the matrix?

B) Suppose that variables i, j, and k are allocated in registers (accessing them does not produce memory references), how many references $N_T = N_W + N_R$ (N_R for reading and N_W for writing data) produces the proposed program (do not consider instruction fetches)?

C) Compute the number of page faults generated by the proposed program. (motivate/justify the answer)

D) By properly completing the following piece of program, it is possible to provide a code equivalent (in terms of behaviour) to the previous one, producing fewer page faults. Complete the program and calculate how many page faults are generated (motivate the answer).

```
#define N 32

data_t M[2*N][N],*V;
int k;
...
V = &(M[0][0]);
for (k=0; k<2*N*N; k++) {
  V[k].diag = ........ /* to be completed */;
}
```

Answers

A) How many pages (and frames) are needed to store the matrix and the array?

N = 32 = 2^5
2*N = 64 = 2^6
2*N*N = 2^11 = 2K
sizeof(data_t) = 1024 = 1K (Bytes)
2KB/1KB = 2, so 1 page contains 2 data_t

|M| = 2*N*N*sizeof(data_t) = 2K * 1KB = 2MB = 2MB/2KB pages = 1K pages = **1024 pages**

The starting address 0x5820C800 is a multiple of the page size (it ends with 11 zeroes), so the matrix starts at a page boundary: no adjustment is needed.

B) Suppose that variables i, j, and k are allocated in registers (accessing them does not produce memory references), how many memory references $N_T = N_W + N_R$ (N_R for reading and N_W for writing data) produces the proposed program (do not consider instruction fetches)?

The total number of iterations of the innermost for is
4*2N*N/4 = 2N = 2K
For each iteration the instruction `M[i][j].diag = i-j;` produces 1 write
N_R = 0
N_w = 2K
N_T = N_R + N_W = 2K

C) Compute the number of page faults generated by the proposed program. (motivate/justify the answer)

No detailed simulation is needed, as (due to the high locality of the program) page faults can be easily estimated.
For each outer iteration (value of k), ¼ of matrix entries are written. Matrix is iterated by rows, writing one entry every 4, so each entry is located within a different page with respect to the previous one.
As the window of 10 pages is not enough to contain ¼ of matrix pages, every access produces a PF.

So the total number of page faults is

N_{PF} = 15394 * 2K/4 = 2K

D) By properly completing the following piece of program, it is possible to provide a code equivalent (in terms of behaviour) to the previous one, producing fewer page faults.

```
#define N 32

data_t M[2*N][N],*V;
int k;
...
V = &(M[0][0]);
for (k=0; k<2*N*N; k++) {
  V[k].diag = ... /* to be completed */;
}
```

Complete the program and compute how many page faults are generated (motivate the answer).

V is a (linear) array view of the matrix. Matrix (row/column) indexes can be computed from linear array indexes as: row = k%N, column = k%N. So the instruction can be competed as follows.

```
V[k].diag = k/N – k%N;
```

As the matrix is now accessed through a single pass/iteration, the number of page faults is now exactly equal to the number of pages.

N_{PF} = 1K
The number of page faults has been halved w.r.t. the original program as every pair of entries in the matrix share a page.

2 Consider a Unix-like file system, based on inodes, with 15 pointers/indexes (**12 direct**, 1 single indirect, 1 double indirect and 1 triple indirect). Pointers/indexes have size 32 bits, and the disk blocks have size 4KB. The file system resides on a disk partition, where 2TB are reserved to data blocks. The extra space reserved for metadata (including index blocks) can be neglected for the purposes of this exercise.

A) It is known that the smallest file in the FS has size 8MB and the largest file has size 8GB. Les's use N2 for the number of files using double indexing and N3 for the number of files using triple indexing. Compute the maximum values (the upper bounds) for N2 and N3.

B) Given a binary file of size 20490.5KB, compute exactly how many index blocks and data blocks the file is using. Also compute the internal fragmentation (for data blocks).

C) Consider the same file of Q B, where the lseek(fd,offset,SEEK_END) operation is called to position the file offset (SEEK_END means that the offset is referred to the end of the file) for the subsequent read/write operation. Suppose fd is the file descriptor associated to the file (already open), and that offset = -2^20. Compute the logical block number (relative to the file blocks, numbered starting at 0) at which the position is moved.

Answers

A) Compute the maximum values (the upper bounds) for N2 and N3.

General observations
An index block contains 4KB/4B = 1K pointers/indexes.
The partition contains 2TB/4KB = 0.5G blocks = 512M blocks

Computing maximum numbers N2/N3 *(for files with* single/triple indirect indexing)
In order to compute the maximum number of files, we need to consider the minimum occupancy. We need to consider both data blocks and index blocks.

Let's use MIN_2 and MIN_3 for minimum occupation of the two kinds of file:

MIN_2 = (12 + 1K + 1) data blocks

But there are two additional constraints:

- it is known that the file sizes have a lower bound of 8MB, which is the size of a file with double indexing (8 MB -> 2K blocks: 12 direct, 1024 single, 1012 double). The occupancy of the file is 2K data blocks.
- it is known that there is at least one file with triple indexing (the largest file of size 8 GB), so he occupancy of the file (2M blocks) has to be subtracted from the space available.

MIN_3 = (12 +1K + 1M +1) data blocks
= 1M + 1037

There is only one additional constraint when computing N3: it is known that at least one file has double indexing, with occupancy 2K blocks (previously computed)

N2 = floor((512M – 2M)/2K) = 255K

N3 = floor((512M-2K) / (1M + 1037)) = 511

B) Given a binary file of size 20490.5KB, compute exactly how many index blocks and data blocks the file is using. Also compute the internal fragmentation (for data blocks).

Data blocks: ceil(20490.5KB/4KB) = 5123

Int. frag = 5123*4KB – 20490.5KB = 1.5KB
alternative method: 20490.5%4 = 0,625 -> int frag = (1-0.625)*4KB = 1.5KB

Index blocks
Single index: 1
Inner index blocks (double): ceil((5123-12-1024)/1024) = 4
Outer index block (double): 1 (double indirect is enough)
Total index blocks: 1+4+1 = 6

C) Consider the same file of Question B. The lseek(fd,offset,SEEK_END) operation is called to position/re-position the file offset (SEEK_END means that the offset is referred to the end of the file) for the subsequent read/write operation. Suppose fd is the file descriptor associated to the file (already open), and that offset = -2^20 (offset is added, relative to SEEK_END, so a negative number is OK). Compute the logical block number (relative to the file blocks, numbered starting at 0) at which the position is moved, and the internal offset (within the block).

Logical address in the file: 20490.5 * 1024 – 1024*1024 = 19466.5*1024 = 19466.5K
Logical block: 19466.5KB / 4KB = 4866
Offset in block: 19466.5KB % 4KB = 0.625 blocks = 0.625*4KB = 2.5KB (as addresses are implicitly expressed in Bytes, the result is 2.5K)

3 A large text file contains N lines of fixed length (50 chars for each line). We need to sort lines in the file using a sorting algorithm of linearithmic complexity (O(NlogN)). Due to the size, sorting is implemented directly on the file (either exploiting memory mapping of files or by properly using seek, read and write primitives.

Depending on the underlying file system (file allocation strategy), which overall complexity do we expect for the sorting task? (for each file allocation strategy, choose the complexity and provide a short motivation)

	O(N)	O(NlogN)	O(N^2)	O(N^2logN)	O(N^3)	Motivation
contiguous						
Linked list						
FAT						
Inodes						

A) A large set of data is stored on file using the strategy called “index and relative file”:
- the index file contains a sequence of N fixed size records, each containing a search key and a pointer (a Logical Address) to the corresponding record in the relative file. The index file is sorted by increasing keys.
- The relative file contains N variable length records. Records are NOT sorted.

Depending on the underlying file system (file allocation strategy), which overall complexity do we expect for the operation of printing all data by increasing order of keys? (for each file allocation strategy, choose the complexity and provide a short motivation)

	O(logN)	O(N)	O(NlogN)	$O(N^2)$	Motivation
contiguous					
Linked list					
FAT					
Inodes					

Answers

A) General comment: the complexity of the algorithm has to be multiplied by the cost of direct/random access to a given LA (Logic Address) within the file.

	O(N)	O(NlogN)	$O(N^2)$	$O(N^2\log N)$	$O(N^3)$	Motivation
contiguous		X				Direct access is supported, so O(1)
Linked list				X		No direct access possible, access is linear (O(N)
FAT				X		Same as with linked lists, but with a much better constant factor.
Inodes		X				Dirrect access supported, so O(1)

B) General comment: printing can follow the order of the index file, so it is linear (O(N)). For each record, the corresponding Logic Address is obtained in constant time (O(1)). Them the complexity of the algorithm has to be multiplied by the cost of direct/random access to a given LA (Logic Address) within the file.

	O(logN)	O(N)	O(NlogN)	$O(N^2)$	Motivation
contiguous		X			Direct access is supported, so O(1)
Linked list				X	No direct access possible, access is linear (O(N)
FAT				X	Same as with linked lists, but with a much better constant factor.
Inodes		X			Dirrect access supported, so O(1)

4 Consider three OS161 kernel threads implementing a data transfer task based on a producer/consumer model (2 producers, 1 consumer). The threads share a data buffer, implemented as a C structure of type `struct prodConsBuf` defined as follows:

```
struct prodConsBuf {
  data_t data;
  int dataReady;
  struct lock *pc_lk;
  struct cv *pc_cv;
};
```

All operations (writing/reading), need mutual exclusion. Whenever no data is present in the buffer (so the buffer is empty), the dataReady flag is 0.
After a producer writes new data to the buffer (data field), he sets dataReady at 1 (so buffer full) and signals the consumer (the work is done by calling function producerWrite).

```
void producerWrite(struct prodConsBuf *pc, data_t *dp);
```

The consumer iteratively executes the consumerRead function.

```
void consumerRead(struct prodConsBuf *pc, data_t *dp);
```

Answer the following questions:

A) Can the shared structure be located into the thread stack, or should it be a global variable or other?

B) As the pc_lk and pc_cv fields are pointers, where
should the lock_create() and cv_create() be called? In the producer thread, in the consumer threads? Elsewhere?

C) The producer and consumer functions are partially given.

```
/* error checking/handling instructions are omitted for simplicity */

void producerWrite(struct prodConsBuf *pc, data_t *dp) {
  /* lock for mutual exclusion */
  lock_acquire(pc->pc_lk);
  /* properly handle here the fact that the buffer could be full
     (filled by the other producer and not yet read by the consumer) */
  <1>
  /* copy data: done here by simple assignment -
     a proper function could be used instead */
  pc->data = *dp;
  pc->dataReady = 1; /* set flag */
  /* signal the consumer */
  <2>
  /* just release the lock and return */
  lock_release(pc->pc_lk);
}

void consumerRead(struct prodConsBuf *pc, data_t *dp) {
  /* lock for mutual exclusion */
  lock_acquire(pc->pc_lk);
  while (!pc->dataReady) {
    cv_wait(pc->pc_cv, pc->pc_lk);
  }
  /* copy data: done here by simple assignment -
     a proper function could be used instead */
  *dp = pc->data;
  pc->dataReady = 0; /* reset set flag */
  /* signal a producer waiting, if any */
  <3>
  /* release the lock and return */
  lock_release(pc->pc_lk);
}
```

Complete their implementation where indicated. Please notice that we need to avoid busy waiting, so a producer signals the consumer when the data is ready. As a producer could be waiting for the consumer to get previous data, before the producer is allowed to write a new data, the consumer needs to do a signal as well.

Answers

A) Can the shared structure be located into the thread stack, or should it be a global variable or other?

> Each thread has its own thread stack, so a shared data cannot reside there. A global variable is the usual option. Other options include dynamic allocation (by kmalloc), if properly handled.

B) As the pc_lk and pc_cv fields are pointers, where should the lock_create() and cv_create() be called? In the producer thread, in the consumer threads? Elsewhere?

It could be done in each of them, provided that they properly synchronize: e.g. one thread does initializations, the other threads wait.

But a more common solution could be that initializations are done by another thread, the parent/master thread, who is creating the producers and the consumer.

C) Complete their implementation where indicated. Please notice that we need to avoid busy waiting, so a producer signals the consumer when the data is ready. As a producer could be waiting for the consumer to get previous data, before the producer is allowed to write a new data, the consumer needs to do a signal as well.

```
<1>
/* wait for the buffer to be empty */
while (pc->dataReady) {
  cv_wait(pc->pc_cv, pc->pc_lk);
}

<2>
/* cv_broadcast is needed as both the consumer and the other producer could
be waiting, in which case both need to be signalled (with cv_signal the
consumer is not guaranteed to be signalled). If we want a more specific
synchronization scheme, we need two cvs */
cv_broadcast(pc->pc_cv, pc->pc_lk);

<3>
/* cv_signal is enough as in the case of two producers waiting, one of them
wil be signalled */
cv_signal(pc->pc_cv, pc->pc_lk);
```

5 Consider a possible implementation of the *open()* and *close()* system calls in OS161.

A) Is the per process table in user memory or in kernel memory?

B) Suppose two user processes, running concurrently, call write(fd,v,N), where fd is 5 for both of them, do you expect that (for each option answer YES/NO and motivate):

1) the processes write on the same file
2) they never write on the same file
3) generally they write on different files but sometimes they can write on the same
4) in order to write on same file, the two processes should use different values for fd.

C) Given the following implementation of sys_write

```
int sys_write(int fd, userptr_t buf_ptr, size_t size) {
  int i;
  char *p = (char *)buf_ptr;

  if (fd!=STDOUT_FILENO && fd!=STDERR_FILENO) {
    return file_write(fd, buf_ptr, size);
  }

  for (i=0; i<(int)size; i++) {
    putch(p[i]);
  }
  return (int)size;
}
```

Assuming that standard input, output and error are mapped to the console (so no redirection to file is possible) can the for loop handling stdout/stderr be replaced by one (or more) of the following code fragments? (For each of them answer yes/no and provide a motivation).

```
1)    for (i=0; i<(int)size; i++) {
          kprintf("%c", p[i]);
      }
2)    kprintf("%s", p);
3)    kprintf("%s", &p);
4)    file_write(fd, p, size);
```

Answers

A)

It is in kernel memory. The table is an internal kernel data structure: the fact of being dedicated to a given process has nothing to do with being accessible in user node to that process.

B)

Preliminary considerations: file descriptors are assigned on a per-process basis and, at the exception of standard input/output/error (but number 5 excludes this case), whenever to processes open a given file, even though the file is the same, there is nothing constraining open to return the same file descriptor. As a consequence, when two processes obtain the same number for a file descriptor, nothing implies that the file is the same.

1) NO. Files can be different.
2) NO. Though non frequent, the file descriptors could be mapped to the same file (of course some locking would be needed)
3) YES. Same explanation as in previous answer.
4) NO. As explained in the preliminary considerations, there is no constrain/bound on file descriptors.

C)

1) YES. Because kprintf, with %c format, is equivalent to putch.
2) NO. Though the C statement is correct, the %s format needs a string terminated by '\0', which cannot be guaranteed by the write system call.
3) NO. In addition to the previous observation on string termination, the statement is formally wrong, as &p is of type char **.
4) NO. Because the file_write function just supports file-based output, not the console. If file_write supported console output, the special case for console handling would not make sense in sys_write.