System and device programming

Iniziato	mercoledì, 26 maggio 2021, 18:15
Stato	Completato
Terminato	mercoledì, 26 maggio 2021, 18:15
Tempo impiegato	8 secondi
Valutazione	**0,00** su un massimo di 15,00 (**0**%)

Domanda 1

Risposta non data

Punteggio max.: 3,00

ALL YES/NO ANSWERS MUST BE EXPLAINED/MOTIVATED. WHEN RESULTS ARE NUMBERS, THE FINAL RESULT, AND RELEVANT INTERMEDIATE STEPS (OR FORMULAS) ARE NEEDED

Consider the following program fragment:

```
#define N 512

float M[2*N][N],V[2*N*N];
int i,j,k,t;
...
for (i=t=0; i<2*N; i++) {
  k = (i<N) ? N-i : i-N+1;
  for (j=0; j<k; j++) {
    V[t++] = M[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:

- the size of a `float` is 32 bits
- the code segment (machine instructions) has size less that one page
- `M` and `V` are allocated at contiguous logical addresses (`M` first, then `V`), starting at logical address 0xA720AC00
- 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 and the array?

B) Suppose now that variable i, j, k and t 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)?

C) Let N_T be the total amount of memory references to data (we omit instruction fetches for sake of simplicity) and N_L be the total number of references to a page that was already accessed within the previous 10 references/accesses. We define a measure of (data) **locality** for the program as the ratio $L = N_L/N_T$. Compute the locality of the proposed program.

D) Compute the number of page faults generated by the proposed program. (motivate the answer)

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

N = 512 = 1/2K
2*N = 1K
N*N = 1/4M = 256K
2KB/4B = 512, so 1 page contains 512 floats

|V| = 2*N*N*sizeof(float) = 2*256K * 4B = 2MB = 2MB/2KB pages = 1K pages = 1024 pages
|M| = |V| = 2MB = 1024 pages
The starting address 0xA720AC00 is NOT a multiple of the page size (it ends with 10 zeroes, whereas it should end with 11 0-bits), it rather starts at 1/2 of a page (C is binary 1100).
We thus need to slightly correct/modify our previous results: V overlaps 1025 pages and M 1025 pages (the first one shared with V): 2049 overall.

B) Suppose now that variable i, j, k and t 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)?

Notation: N_i number of iterations of the outer for
N_j number of iterations of the inner for
Solution

```
for (i=0;i<2*N;i++){ // Ni = 2*N = 1K iterations
// for the first N iterations of i, k goes from N-1 down to 0
// for the next N iterations of i, k goes from 0 up to N-1
for (j=0;j<=k;j++){ the average k value is (N+1)/2, repeated for all i values
// Nj = 2*S_{k=0..N-1}(k+1) = 2N(N+1)/2 = N(N+1) iterations,
```

N_j = 512*513 = 256,5K = 262656

`V[t++] = M[i][j]; //` **N_R = N_j reads, N_w = N_j writes, $N_T = N_R + N_W = 2*N_j$ = 513K**

Alternative (quick) solution.

The proposed matrix is a rectangular matrix that can be viewed as the union of two square matrices of size `N*N`. Due to the way `k` is computed, it has an average value of `(N+1)/2`, so the average number of `j` iterations for each value of `i` is `(N+1)/2`. Overall, the number of iterations of the inner loop is `2N*(N+1)/2`, with one read and one write for each iteration. So **N_j = N*(N+1) = 512*513, $N_R = N_w = N_j$** (one read + one write per iteration). **$N_T = N_R + N_W = 2*N_j$**

C) Let N_T be the total amount of memory references to data (we omit instruction fetches for sake of simplicity) and N_L be the total number of references to a page that was already accessed within the previous 10 references/accesses. We define a measure of (data) **locality** for the program as the ratio $L = N_L/N_T$. Compute the locality of the proposed program.

Each row of M overlaps two pages, as the first page only contains half of the first row, and the last row is shared with V. Not all cells of M are read, but all pages of M are read at least than once: the only non local access being the first one to that page.
V is written sequentially, but not fully, just for the first N_j cells, with N_j = `512*513` (as previously computed). V shares (half and half, so 1KB = 256 floats) the first page with M. The remaining pages amount to *ceil*((N_j – 256)/512) = *ceil*(512,5) = 513 pages
The first part ov V is in the same page as the last page of M.
For all accessed pages, we just have one non local access (the page shared by M and V has 2 non local accesses, one when writing V and one when reading M)

N_{NL} = 1025 (M pages) + 1 (V page shared with M) + 513 (other V pages) = 1539

$N_L = N_T – N_{NL}$
$L = N_L/N_T = (N_T – N_{NL})/N_T = 1 – N_{NL}/N_T = 1 – 1539/1024*513 = 1 – 0.0029 = 0,9969$

D) Compute the number of page faults generated by the proposed program. (motivate the answer)

No detailed simulation is needed, as (due to the high locality of the program) page faults can be easily estimated.

There is exactly one page fault for each non local access so N_{PF} = 1539

Domanda 2

Risposta non data

Punteggio max.: 3,00

ALL YES/NO ANSWERS MUST BE EXPLAINED/MOTIVATED. WHEN RESULTS ARE NUMBERS, THE FINAL RESULT, AND RELEVANT INTERMEDIATE STEPS (OR FORMULAS) ARE NEEDED

A given disk is organized with physical and logical blocks of size 8KB. The disk contains multiple partitions: partition A, of NB blocks, is formatted for a file system that statically allocates NM blocks for metadata (that include directories, file control blocks and a bitmap, for fee space management), and ND blocks for file data. The bitmap has one bit for each of the ND data blocks. NM/2 of the metadata blocks are reserved to the bitmap.

Answer the following questions:

A) Compute the ratio ND/NM

B) Suppose the bitmap indicates a 25% ratio free/used blocks (so 1 free block for 4 used), compute (as a function of NM) the maximum size for a contiguous interval of free blocks, assuming the most favorable bitmap configuration. Give the same answer when assuming the less favorable bitmap configuration.

C) We know that a file control block (FCB) has size 256B and NM/4 metadata blocks are reserved to FCBs, for a maximum of 16K files. Compute ND, NM and NB. Also express the size of the bitmap and of the A partition, expressed in Bytes.

A) Compute the ratio ND/NM

|bitmap| = NM/2 blocks = NM/2*8K*8 bits = 32K*NM bits
each bitmap bit corresponds to one of the ND data blocks
ND = 32K*NM
ND/NM = 32K

B) Suppose the bitmap indicates a 25% ratio free/used blocks (so 1 free block for 4 used), compute (as a function of NM) the maximum size for a contiguous interval of free blocks, assuming the most favorable bitmap configuration. Give the same answer when assuming the less favorable bitmap configuration.

N_{free}/N_{used} = 0,25 => N_{free} = 0.20*(N_{free}+N_{used}) = 0.2*(N_{bits}) = 0.2*32K*NM bits = 6.4K*NM bits
The most favourable situation is when all free blocks are contiguous:
N_{good} = 6.4K*NM

The less favourable situation is when all free blocks have no other adjacent free block:
N_{bad} = 1

C) We know that a file control block (FCB) has size 256B and NM/4 metadata blocks are reserved to FCBs, for a maximum of 16K files. Compute ND, NM and NB. Also express the size of the bitmap and of the A partition, expressed in Bytes.

The maximum number of files is 16K
A block can contain 8KB/256B = 32 FCBs
The maximum number of FCBs is 32*NM/4 = 8*NM
As files correspond to FCBs:
8*NM = 16K
NM = 2K
ND = 32K*NM = 64M
NB = ND + NM = 64M + 2K
|A| = (64M+2K)*8KB = 512GB + 16MB
|bitmap| = NM/2*8KB = 8MB

Domanda 3

Risposta non data

Punteggio max.: 3,00

ALL YES/NO ANSWERS MUST BE EXPLAINED/MOTIVATED. WHEN RESULTS ARE NUMBERS, THE FINAL RESULT, AND RELEVANT INTERMEDIATE STEPS (OR FORMULAS) ARE NEEDED

Answer the following questions on memory management:

A) Consider dynamic loading and dynamic linking. Can a program be dynamically loaded without the support of dynamic linking? Does dynamic linking require a dynamically loadable program?

B) Briefly explain why an Inverted Page Table needs a pid (process id) field in each of its entries, whereas this is not true for a standard page table.

C) Consider a CPU equipped with a TLB, can the TLB contain entries of multiple processes or is it constrained to contain entries for just one process?

D) Given a 64 bit CPU, handling physical addresses of just 42 bits, and hierarchical paging, 64 bit logical addresses split in (p1,p2,d), of size (40,12,12) bits, respectively, compute the minimum number of bits needed by a TLB entry, considering that each TLB entry includes "validity" and "modify" bits, and 3 page protection bits.

A) Consider dynamic loading and dynamic linking.

Can a program be dynamically loaded without the support of dynamic linking?

Yes. Although dynamic linking can be combined to dynamic loading, it is not mandatory: a program can be fully linked and loaded dynamically/incrementally, e.g. program-controlled dynamic loading

Note: dynamic loading means that parts of a program are loaded in memory just when needed, dynamic linking means that inter-module references are resolved at run time (it is particularly useful for libraries).

Does dynamic linking require a dynamically loadable program?

No. Again, Although dynamic loading can exploit dynamic linking, a program can be dynamically linked (e.g. to a shared library already loaded in memory) without dynamic loading

B) Briefly explain why an Inverted Page Table needs a pid (process id) field in each of its entries, whereas this is not true for a standard page table.

Because the IPT includes “by construction” page/frame entries for all processes in a system, as the IPT is mapped to the physical memory, NOT to a given process. So each entry needs a pid, as different processes can have same p (page) values, obviously mapped to different frames. On the other hand, a standard page table is typically a per-process table, so it doesn’t need pid values. If a standard page table is a system-wide PT, then it also needs a pid field in each entry.

C) Consider a CPU equipped with a TLB, can the TLB contain entries of multiple processes or is it constrained to contain entries for just one process?

Both types of TLBs exist: the ones containing entries for multiple processes and the one containing just entries for the currently active process. The latter ones need a TLB cleanup/reset at each context switch

D) Given a 64 bit CPU, handling physical addresses of just 42 bits, and hierarchical paging, 64 bit logical addresses split in (p1,p2,d), of size (40,12,12) bits, respectively, compute the minimum number of bits needed by a TLB entry, considering that each TLB entry includes “validity” and “modify” bits, and 3 page protection bits.

A TLB entry has (p,f), i.e. page (including both p1 and p2), frame, plus additional bits (5 are needed)
|f| is 42-12 (offset removed from physical address)
|TLB-entry| = 40+12+30+5 = 87 bits

Domanda 4

Risposta non data

Punteggio max.: 3,00

ALL YES/NO ANSWERS MUST BE EXPLAINED/MOTIVATED. WHEN RESULTS ARE NUMBERS, THE FINAL RESULT, AND RELEVANT INTERMEDIATE STEPS (OR FORMULAS) ARE NEEDED

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

```
#define NumP 2

struct prodCons {
  int cnt[NumP];
  int size[NumP];
  struct lock *pc_lk;
  struct cv *pc_cv;
  … /* data buffer handling – omitted */
};
```

Producer `i` (with `i = 0` or `1`) updates the `i`-th entries in the `cnt` and `size` arrays. The consumer reads both arrays. The lock is used for mutual exclusion on shared arrays. Before working on data, the consumer needs to wait for a condition to be true: that either `(cnt[0]+cnt[1]) > minCnt` or that `(size[0]+size[1]) > minSize` *(though `cnt` and `size` are fields of a C struct, prefix structure name has been omitted here for simplicity).* This is done by calling function

```
void consumerWait(struct prodCons *pc, int minCnt, int minSize);
```

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 threads, in the consumer thread? Elsewhere?

C) Provide an implementation of function `consumerWait`. No producer and/or consumer code is needed, just the function.

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 threads, in the consumer thread? 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) Provide an implementation of function `consumerWait`. No producer and/or consumer code is needed, just the function.

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

void consumerWait(struct prodCons *pc, int minCnt, int
minSize) {
  /* lock for mutual exclusion (producers will use lock too)
*/
  lock_acquire(pc->pc_lk);
  /* a producer can call cv_signal after any modification to
cnt/size fields
     or just when the modification reaches the condition
needed by the
     consumer. The while cycle is needed in both cases (due
to mesa semantics
   */
  while ((pc->cnt[0]+pc->cnt[1])<=minCnt &&
(pc->size[0]+pc->size[1])<=minSize) {
    cv_wait(pc->pc_cv, pc->pc_lk);
  }
  lock_release(pc->pc_lk);
}
```

Domanda 5

Risposta non data

Punteggio max.: 3,00

ALL YES/NO ANSWERS MUST BE EXPLAINED/MOTIVATED. WHEN RESULTS ARE NUMBERS, THE FINAL RESULT, AND RELEVANT INTERMEDIATE STEPS (OR FORMULAS) ARE NEEDED

Consider a possible implementation of the *open()* and *close()* system calls in OS161. The *open()* system call is implemented by a function sys_open(), which internally has the following instruction:

```
err = vfs_open(name, flags, &vn);
```

A) Is `vn` (standing for vnode) a pointer or a C structure?

B) Suppose two user processes call open() for the same file, do you expect (select an option and motivate):

1 – each vfs_open to create a new vnode, copied in vn

2 – each vfs_open to create a new vnode, whose pointer is returned in vn

3 – just one vnode is created and copied in vn (so we have two copies)

4 – just one vnode is created and the pointer to it returned in vn (so we have two pointers to the same vnode)

C) The system open file table is implemented as follows

```
/* system open file table */
struct openfile {
  struct vnode *vn;
  off_t offset;
  unsigned int countRef;
};

struct openfile systemFileTable[SYSTEM_OPEN_MAX];
```

Why does `struct openfile` include a counter of references (countRef), while vnodes already have internally a counter of references? Is this a duplicate?
Is it possible for a given entry in the `systemFileTable` array to be shared (pointed) by two processes?

A) Is `vn` (standing for vnode) a pointer or a C structure?

It is a pointer, to be declared as
struct vnode *vn;
It cannot be a structure as the vfs_open returns (by pointer, so the &) a pointer to a vnode, allocated in a dedicated table, not a copy of the vnode

B) Suppose two user processes call open() for the same file, do you expect (select an option and motivate):

1 – each vfs_open to create a new vnode, copied in vn

2 – each vfs_open to create a new vnode, whose pointer is returned in vn

3 – just one vnode is created and copied in vn (so we have two copies)

4 – just one vnode is created and the pointer to it returned in vn (so we have two pointers to the same vnode)

Number 4, as vfs_open returns a pointer and not a copy (so 1 and 3 are excluded). 2 is also excluded as just one vnode is possible for a given file. If the file is opened multiple times (concurrently) a vnode will be created just once, and referenced in all next open.

C) The system open file table is implemented as follows

```
/* system open file table */
struct openfile {
  struct vnode *vn;
  off_t offset;
  unsigned int countRef;
};

struct openfile systemFileTable[SYSTEM_OPEN_MAX];
```

Why does `struct openfile` include a counter of references (countRef), while vnodes already have internally a counter of references? Is this a duplicate?

Because both a `vnode` and a `struct openfile` can be shared, in different ways: a vnode can be shared by multiple entries in `systemFileTable`. A `struct openfile` can be shared by multiple entries in process file tables, for the same shared file.

Is it possible for a given entry in the `systemFileTable` array to be shared (pointed) by two processes?

Yes. This is for instance a case happening within the fork of a process, when (right after forking) the process file tables of the parent and child processes, which are two distinct tables, point to shared entries, with countRef fields properly incremented during fork