Rešenja zadataka za kolokvijum iz Operativnih sistema 1 - Rok 5/2025.

1. (10 poena)

Svaki PMT zauzima tačno jednu stranicu veličine 4 KB = 2^{12} B sa ulazima veličine 8 B = 2^3 B, pa poseduje $2^{12}/2^3 = 2^9$ ulaza za čije adresiranje je potrebno 9 bita. Polje za pomeraj unutar virtuelne adrese je 12 bita, pa preostalih 48-12 = 36 bita adresira stranicu. Adresiranje najnižeg 1 GB virtuelnog adresnog prostora zahteva 30 bita, od kojih je 12 za pomeraj unutar stranice, a preostalih 18 za adresiranje stranice.

a)(5) Za 36 bita za broj stranice i po 9 bita za svako polje za adresiranje ulaza unutar PMT-a svakog nivoa, potrebno je 36/9 = 4 nivoa PMT-a. Za adresiranje 1 GB kernel prostora koji se deli između procesa potrebno je i dovoljna jedna stranica za PMT 3. nivoa i 2^9 = 512 stranica za PMT-ove 4. nivoa, što je ukupno 513 stranica. Za adresiranje tog prostora i još jedne stranice iznad njega, za svaki proces potrebno je alocirati po jedan PMT sva 4 nivoa. Prema tome, ukupno je za sve PMT-ove n procesa potrebno $4n + 513$ stranica.

b)(5) Za PMT-ove za adresiranje kernel prostora potrebno je istih 513 stranica za PMT-ove u dva nivoa koje svi procesi dele kao pod a). Za adresiranje ostatka virtuelnog adresnog prostora potrebna je ista organizacija PMT u 4 nivoa kao pod a), samo što se ulazi 0 u PMT-ovima 1. i 2. nivoa praktično ne koriste (ne može se iskoristiti za smanjenje broja nivoa PMT-a). Kada bi svi procesi adresirali samo kernel prostor, ukupno bi bilo potrebno 515 stranica za PMT-ove koji se svi dele za proizvoljan broj procesa. Međutim, ako proces alocira još jednu stranicu iznad tog prostora, za svaki proces potrebna je po jedna stranica za sva 4 nivoa PMT-a, pa je odgovor isti kao pod a).

Zaključak: od ovakve organizacije opisane pod b) nema nikakve posebne dobiti u smislu zauzeća memorije za PMT-ove. Ona se praktično svodi na to da su ulazi 0 u PMT-u 1. i 2. nivoa svih procesa praktično keširani u PMTPX i deljeni, dok bi kod organizacije pod a) TLB morao da kešira sve pojedinačne deskriptore za različite procese iako ukazuju na iste stranice, pa bi se mogao očekivati bolji učinak u smislu performansi i korišćenja TLB-a.

2. (10 poena)

```
class Event {
public:
  Event ();

  int wait();
  void signal();
```

```
private:
  Thread* owner;
  short val;
};

inline Event::Event () : val(0) {
  this->owner = Thread::runningThread;
}

int Event::wait () {
  if (this->owner != Thread::runningThread) return -1;
  lock();
  Thread* oldRunning = Thread::runningThread;
  if (val-- == 1)
    Scheduler::put(oldRunning);
  Thread* newRunning = Thread::runningThread = Scheduler::get();
  if (oldRunning!=newRunning)
    Thread::yield(oldRunning,newRunning);
  unlock();
  return 0;
}

void Semaphore::signal () {
  lock();
  Thread* oldRunning = Thread::runningThread;
  if (val++ == -1)
    Scheduler::put(this->owner);
  else
    val = 1;
  Scheduler::put(oldRunning);
  Thread* newRunning = Thread::runningThread = Scheduler::get();
  if (oldRunning!=newRunning)
    Thread::yield(oldRunning,newRunning);
  unlock();
}
```

3. (10 poena)

```
IORequest *dmaPending[NDMAs] = {}; // Currently pending requests

const REG DMA_TRANSFER_COMPLETE = 1, DMA_ERROR = 2, DMA_START = 1;

void startDMA (int i) {
  if (ioHead!=0 && dmaPending[i]==0) {
    dmaPending[i] = ioHead; // Take the first request,
    ioHead = ioHead->next; // remove it from the list
    dmaAddress[i] = (REG)dmaPending[i]->buffer; // and assign it to DMA[i]
    dma1Count[i] = (REG)dmaPending[i]->size;
    dma1Ctrl[i] = DMA_START; // Start I/O
  }
}

void transfer () {
  for (int i=0; i<NDMAs; i++) startDMA(i);
}
```

```
interrupt void dmaInterrupt () {
  for (int i=0; i<NDMAs; i++)
    if (dmaStatus[i] & DMA_TRANSFER_COMPLETE) { // DMA[i] completed
      if (dmaPending[i]==0) continue; // Exception
      if (dmaStatus[i] & DMA_ERROR) // Error in I/O
        dmaPending[i]->status = -1;
      else
        dmaPending[i]->status = 0;
      dmaPending[i] = 0;
      startDMA(i);
    }
}
```

4. (10 poena)

```
const int FNAME_LEN = 8, FEXT_LEN = 3;
const int NUM_DENTRIES = 512/32;
const char DENTRY_EM PTY = '\0xE5';

int strcmp (const char* str1, const char* str2, int len) {
  for (int i=0; i<len; i++)
    if (*str1++ != *str2++) return 0;
  return 1;
}

FATDirEntry* getEntry (FATDirEntry* dir,
                  const char name[FNAME_LEN], const char ext[FEXT_LEN]) {
  uint16_t cluster = dir->firstCluster;
  while (cluster) {
    FATDirEntry* entry = (FATDirEntry*)FSCache::getCluster(cluster);
    for (int i=0; i<NUM_DENTRIES && entry->name[0]; i++, entry++) {
      if (entry->name[0]!=DENTRY_EMPTY &&
          strcmp(entry->name,name,FNAME_LEN) &&
          strcmp(entry->ext,ext,FNAME_EXT)) return entry;
    }
    cluster = FAT::getNextCluster(cluster);
  }
  return 0;
```