گزارش کار پروژه کامپیوتری سوم

سؤالات تشریحی

۱. در سیستم عامل 6xv، بلوک کنترل فرآیند (Process Control Block یا PCB) با استفاده از ساختار struct proc پیاده‌سازی شده است.این ساختار تمام اطلاعات لازم برای مدیریت یک فرآیند را نگهداری می‌کند. همچنین، وضعیت‌های فرآیند با استفاده از enumeration به نام enum procstate تعریف شده‌اند.

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

وضعیت‌های فرآیند در 6xv به شرح زیر هستند:

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```

شکل 3.3 معمولاً یک PCB عمومی را نشان می‌دهد که شامل وضعیت فرآیند، PID، کانتر برنامه، رجیسترها، اطلاعات حافظه، فایل‌های باز و سایر داده‌های کرنلی است. 6xv این عناصر را به صورت مشابه پیاده‌سازی کرده، اما با جزئیات خاص معماری 86x و سادگی آموزشی.

ساختار 6xv بسیار نزدیک به PCB استاندارد است و عناصر کلیدی مانند وضعیت، شناسه، رجیسترها، حافظه و فایل‌ها را پوشش می‌دهد. تفاوت‌ها بیشتر در جزئیات پیاده‌سازی (مانند استفاده از pgdir برای صفحه‌بندی به جای محدودیت‌های ساده حافظه) یا فیلدهای اضافی برای همگام‌سازی (مانند chan) است که به دلیل سادگی 6xv اضافه شده‌اند. این شباهت‌ها نشان‌دهنده پیروی 6xv از اصول طراحی سیستم‌عامل‌های یونیکس‌مانند است.

2.

وضعیت در xv6	معادل در شکل ۱	توضیح مختصر
UNUSED	(وجود ندارد)	این وضعیت در 6xv برای اسلات خالی جدول proc است.
EMBRYO	new	فرآیند تازه ایجاد شده، هنوز به حالت ready نرفته
RUNNABLE	ready	فرآیند کاملاً آماده اجراست و در صف scheduler (ready queue) قرار دارد.
RUNNING	running	فرآیند در حال حاضر روی CPU در حال اجراست
SLEEPING	waiting	منتظر I/O یا رویداد
ZOMBIE	terminated	پایان یافته، منتظر جمع‌آوری توسط والد

6xv دقیقاً همان مدل ۵ حالته استاندارد را پیاده‌سازی کرده است، فقط نام دو حالت میانی را کمی متفاوت انتخاب کرده (EMBRYO به جای new و ZOMBIE به جای terminated)، اما رفتار و انتقال بین حالت‌ها کاملاً مطابق شکل ۱ است.

حالت UNUSED فقط یک وضعیت داخلی برای مدیریت جدول فرآیندها است و جزو چرخه حیات واقعی فرآیند محسوب نمی‌شود.

در سیستم‌عامل 6xv، حالت «new» در شکل استاندارد ۵ حالته معادل وضعیت EMBRYO و حالت «ready» معادل وضعیت RUNNABLE است.

گذار از حالت **new** → **ready** تنها زمانی رخ می‌دهد که یک فرآیند جدید کاملاً ایجاد و آماده‌سازی شده و آماده قرار گرفتن در صف زمان‌بندی (ready queue) شود.

دو تابعی که در فایل **proc.c** مستقیماً مسئول این گذار هستند، عبارتند از:

تابع fork

این تابع برای ایجاد فرآیند فرزند استفاده می‌شود. ابتدا تابع allocproc() را فراخوانی می‌کند که یک اسلات خالی در جدول فرآیندها پیدا کرده و وضعیت آن را به EMBRYO تغییر می‌دهد. پس از تکمیل تمام مراحل کپی کردن فضای آدرس، فایل‌های باز، تنظیم trapframe و ...، در انتهای تابع fork() وضعیت فرآیند فرزند به صورت زیر به RUNNABLE تغییر می‌کند.

تابع userinit

این تابع برای ایجاد اولین فرآیند کاربر (initprocess) فراخوانی می‌شود. مشابه fork، ابتدا allocproc() وضعیت را به EMBRYO می‌برد و پس از تنظیم فضای آدرس، پشته کاربر، trapframe و ...، در انتهای تابع وضعیت را به RUNNABLE تغییر می‌دهد.

۴.

حداکثر تعداد فرآیندهای همزمان در 6xv برابر با ۶۴ است .

اگر یک برنامه سعی کند بیش از ۶۴ فرآیند ایجاد کند، تابع allocproc نمی‌تواند اسلات خالی پیدا کند و مقدار ۰ برمی‌گرداند. در نتیجه تابع fork مقدار ۱- به برنامهٔ سطح کاربر برمی‌گرداند و فرآیند جدیدی ایجاد نمی‌شود. برنامهٔ کاربر فقط شکست فراخوانی fork را مشاهده می‌کند و کرنل هیچ panic یا خطای دیگری تولید نمی‌کند؛ صرفاً ایجاد فرآیند جدید با شکست مواجه می‌شود.

۵.

در تابع scheduler، جدول فرآیندها (ptable) در ابتدای هر حلقه قفل می‌شود تا از دسترسی همزمان و ناسازگار به جدول فرآیندها جلوگیری شود. این قفل حتی در سیستم‌های تک‌هسته‌ای نیز ضروری است، زیرا وقفه‌های ساعت و سیستم‌کال‌ها می‌توانند در هر لحظه وضعیت فرآیندها را تغییر دهند و بدون قفل، race condition رخ می‌دهد. در سیستم‌های چندهسته‌ای این نیاز دوچندان است، زیرا چندین scheduler به‌طور همزمان روی هسته‌های مختلف اجرا می‌شوند. بنابراین قفل کردن جدول فرآیندها در هر چرخهٔ scheduler برای تضمین صحت و پایداری سیستم عامل 6xv کاملاً اجباری است.

۶.

در مکانیزم زمان‌بندی 6xv، به دلیل وجود قفل واحد ptable.lock روی کل جدول فرآیندها، وقتی یک هسته در حال پیمایش جدول (خط ۳۳۵ proc.c) است، هیچ هستهٔ دیگری نمی‌تواند وضعیت فرآیندها را تغییر دهد. بنابراین اگر یک فرآیند در حین پیمایش یک هسته به حالت RUNNABLE درآید، این تغییر تنها پس از آزاد شدن قفل و در iteration بعدی رخ می‌دهد. در نتیجه، فرآیند تازه آماده‌شده **در همان iteration جاری دیده نمی‌شود** و تنها در **iteration بعدی** حلقهٔ scheduler (پس از یک دور کامل حلقه و گرفتن دوباره قفل توسط هسته‌ها) امکان زمان‌بندی و انتخاب شدن را پیدا می‌کند.

این رفتار ناشی از طراحی ساده و استفاده از یک قفل بزرگ (big kernel lock) روی جدول فرآیندها در 6xv است.

۷.

```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

۸.

رجیستر Program Counter در ساختار context در نسخهٔ 86x به نام eip و در نسخهٔ RISC-V به نام ra شناخته می‌شود. ذخیرهٔ آن هنگام خروج از فرآیند به scheduler توسط دستور call swtch و کد اسمبلی تابع swtch انجام می‌شود که مقدار eip/ra را از پشته خوانده و در context ذخیره می‌کند. بازنشانی آن هنگام بازگشت از scheduler به فرآیند توسط دستور ret در انتهای swtch انجام می‌شود که مقدار ذخیره‌شده در context->eip یا context->ra را دوباره در رجیستر PC بارگذاری کرده و اجرای کد کرنل فرآیند را دقیقاً از همان نقطهٔ قبلی ادامه می‌دهد.

۹.

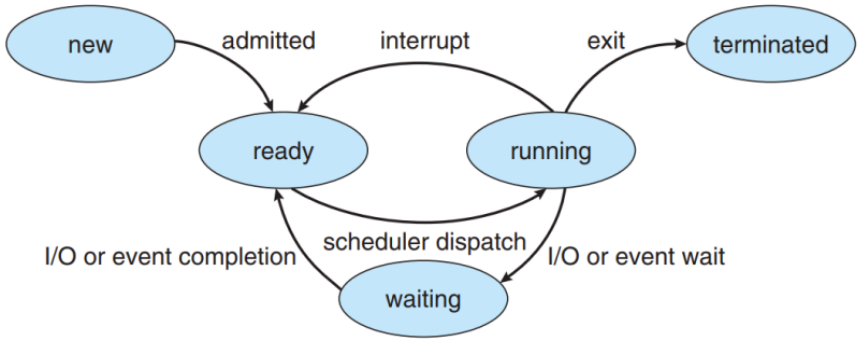
اگر در ابتدای scheduler دستور sti وجود نداشته باشد و وقفه‌ها غیرفعال بمانند، وقفهٔ تایمر (و هیچ وقفهٔ دیگری) نمی‌تواند اجرا شود، زیرا CPU تا وقتی پرچم IF=0 باشد وقفه‌های خارجی را نادیده می‌گیرد. در نتیجه تابع yield() هرگز فراخوانی نمی‌شود، هیچ فرآیندی CPU را رها نمی‌کند و scheduler در حلقهٔ بی‌نهایت خود بدون انجام سوئیچ واقعی باقی می‌ماند. این وضعیت باعث توقف کامل هسته (و در نهایت کل سیستم) می‌شود. بنابراین فعال کردن وقفه‌ها با sti در ابتدای scheduler کاملاً ضروری است تا وقفهٔ تایمر بتواند در هر لحظه رخ دهد و امکان زمان‌بندی پیش‌گیرانه (preemptive scheduling) فراهم شود.

۱۰.

```
lapicw(TDCR, X1);
lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
lapicw(TICR, 10000000);
```

با توجه به تنظیمات تایمر در کد 6xv، مقدار TICR برابر 10000000 قرار داده شده است و تقسیم‌کنندهٔ تایمر (TDCR) نیز روی ۱ تنظیم شده است. با فرض فرکانس ورودی ۱۰۰MHz، تایمر هر ۱۰,۰۰۰,۰۰۰ سیکل یک بار به صفر می‌رسد و وقفهٔ دوره‌ای تولید می‌کند. در نتیجه وقفهٔ تایمر تقریباً هر **۰٫۱ ثانیه (۱۰۰ms)** صادر می‌شود. این مقدار دقیقاً همان کوانتوم زمانی زمان‌بندی نوبت‌گردشی در 6xv است.

۱۱.



شکل ۱. چرخهٔ وضعیت یک پردازه.

تابعی که در 6xv منجر به انجام گذار interrupt در شکل استاندارد می‌شود، تابع yield است. این تابع هم در اثر وقفهٔ تایمر و هم در اثر فراخوانی توابع مسدودکننده مثل sleep صدا زده می‌شود و باعث می‌شود فرآیند جاری CPU را رها کند و کنترل به scheduler برگردد.

۱۲.کوانتوم زمانی سیستم زمان‌بندی 6xv برابر **حدود ۱۰۰ میلی‌ثانیه** است؛ زیرا تنها رخداد فعال‌کنندهٔ تعویض نوبت، **interrupt تایمر** است که با نرخ ۱۰ هرتز پیکربندی شده است. در نتیجه هر پردازه پس از حدود ms۱۰۰ اجرای پیاپی، نوبت را به پردازهٔ بعدی واگذار می‌کند.

۱۳.

در فایل proc.c تابع wait ابتدا تمام فرآیندهای موجود در جدول فرآیندها را بررسی می‌کند تا فرآیندهای زامبی را پیدا کند. اگر چنین فرزندی پیدا شد، منابع آن آزاد شده و بلافاصله مقدار خروج آن برگردانده می‌شود. اما اگر هیچ فرزند زامبی وجود نداشت فرآیند والد خودش را در وضعیت SLEEPING قرار می‌دهد و با فراخوانی تابع sleep روی آدرس فرآیند خود (p->parent) منتظر می‌ماند تا یکی از فرزندانش تمام شود. وقتی یک فرزند با exit خارج می‌شود، در تابع exit با فراخوانی wakeup والد را بیدار می‌کند. سپس والد ازsleep برمی‌گردد و دوباره حلقه را ادامه می‌دهد تا فرزند زامبی را جمع‌آوری کند.

---

۱۴. انتظار برای آزاد شدن حافظه در تخصیص‌دهندهٔ هسته:

وقتی یک فرآیند یا هسته نیاز به تخصیص یک صفحهٔ فیزیکی جدید دارد، تابع kalloc فراخوانی می‌شود. اگر در آن لحظه هیچ صفحهٔ آزادی در لیست freelist وجود نداشته باشد، به‌جای برگرداندن خطا، هسته ترجیح می‌دهد صبر کند تا حافظه آزاد شود. به این ترتیب، فرآیند جاری روی قفل مدیریت حافظه می‌خوابد. هر وقت فرآیند دیگری با kfree صفحه‌ای را آزاد کند همهٔ فرآیندهایی که در kalloc خوابیده بودند بیدار می‌شوند و شانس دوباره گرفتن حافظه را دارن

۱۵.الف) تابع wakeup مسئول آگاه‌سازی فرآیندهایی است که با sleep() خوابیده‌اند. این تابع یک آرگومان می‌گیرد و تمام فرآیندهایی که روی همان channel خوابیده‌اند را از وضعیت SLEEPING به وضعیت RUNNABLE منتقل می‌کند.

ب) تابع wakeup باعث گذار وضعیت فرآیند از SLEEPING → RUNNABLE می‌شود.

ج) بله، علاوه بر wakeup دو تابع دیگر هم می‌توانند یک فرآیند خوابیده را مستقیماً به وضعیت RUNNABLE ببرند یک فرآیند در حالت SLEEPING اگر سیگنال kill بگیرد، **از خواب بلند می‌شود** و **RUNNABLE** می‌شود.exit باعث می‌شود فرآیند به حالت **زومبی** برود تا پدرش با wait آن را جمع کند.

۱۶.

در سیستم‌عامل 6xv هیچ فرآیند orphanای وجود ندارد. وقتی یک فرآیند با exit خاتمه می‌یابد، اگر والد آن قبلاً مرده باشد، 6xv به طور خودکار والد تمام فرزندان آن فرآیند را به فرآیند init (با pid برابر 1) تغییر می‌دهد. سپس فرآیند init به صورت دوره‌ای با فراخوانی wait وضعیت خروج این فرآیندهای zombie را خوانده و منابع آن‌ها را آزاد می‌کند. این مکانیسم باعث می‌شود که هیچ‌گاه فرآیند یتیمی در سیستم باقی نماند و مشکل zombieها نیز به طور کامل مدیریت شود.

---

۱۷. در سیستم‌عامل 6xv اطلاعات مربوط به هر CPU در ساختاری به نام struct cpu نگهداری می‌شود که در فایل param.h و proc.h تعریف شده است. این ساختار به ازای هر هستهٔ پردازنده یکی وجود دارد و آرایه‌ای از آن با نام cpus در حافظه قرار دارد.

```
struct cpu {
  uchar apicid;                // Local APIC ID
  struct context *scheduler;   // swtch() here to enter scheduler
  struct taskstate ts;         // Used by x86 to find stack for interrupt
  struct segdesc gdt[NSEGS];   // x86 global descriptor table
  volatile uint started;       // Has the CPU started?
  int ncli;                    // Depth of pushcli nesting.
  int intena;                  // Were interrupts enabled before pushcli?
  struct proc *proc;           // The process running on this cpu or null
};
```

struct proc *proc

اشاره‌گری به فرایند جاری است؛

struct context context

این ساختار مجموعه‌ای از ثبات‌های مهم پردازنده را نگه می‌دارد. این مقادیر زمانی ذخیره می‌شوند که scheduler از CPU خارج می‌شود و قرار است بین scheduler و یک فرایند جابه‌جایی انجام شود.

int noff

شمارندهٔ میزان تو در تو بودن غیرفعال‌سازی وقفه‌هاست. هر بار که تابع **push_off** فراخوانی شود این مقدار افزایش می‌یابد و با **pop_off**کاهش پیدا می‌کند. تا زمانی که مقدار **noff > 0** باشد، وقفه‌ها دوباره فعال نخواهند شد.

int intena

نشان می‌دهد که قبل از آخرین باری که CPU وارد scheduler شده، وقفه‌ها فعال بوده‌اند یا خیر. این مقدار به scheduler کمک می‌کند پس از بازگشت به اجرای فرایند، وضعیت وقفه‌ها را به درستی به حالت قبلی برگرداند.

uchar started

مشخص می‌کند که آیا این هستهٔ پردازشی بوت شده و scheduler روی آن شروع به کار کرده است یا نه.