بسمه تعالی

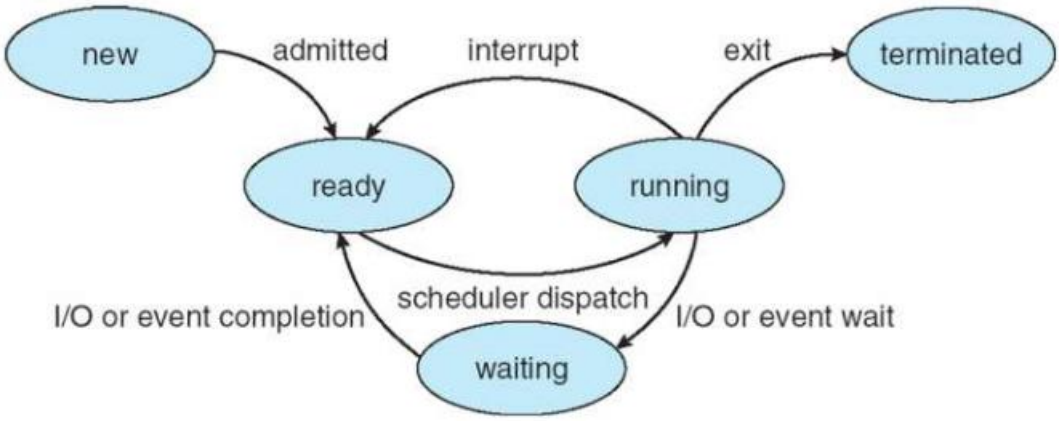
تمرین شماره ۲

سیستم‌های عامل

محسن کربلائی امینی، ۹۸۲۴۲۱۲۸

آبان ۱۴۰۲

سوال ۱:



با توجه به تصویر بالا:

الف)
در صورت تایید شدن توسط long-term scheduler به حالت ready می‌رود.

ب)
در صورت تایید شدن توسط short-term scheduler به حالت running می‌رود.

پ)
در صورت دریافت وقفه به حالت ready و در صورت درخواست منابع و یا event به حالت waiting می‌رود.

ت)
در صورت تایید درخواست منابع خواسته شده به حالت ready می‌رود.

سوال ۲:
در صورت کاهش نرخ context switch، همزمانی در سیستم کاهش می‌یابد و هر پراسس باید برای پردازش منتظر اتمام پردازش پراسس‌های دیگر باشد. مثلا ممکن است هیچ وقت نوبت به پراسس حرکت موس نرسد و با چند ثانیه تاخیر اجرا شود.
در صورت افزایش این نرخ، سربار ناشی از پراسس خود کرنل برای جابه‌جایی ایجاد می‌شود و زمان کمتری به خود پراسس‌ها می‌رسد. در نتیجه بهره‌وری سیستم کاهش می‌یابد.

سوال ۳:

به صورت کلی می‌توان ایراد به مکانیزم shared memory را این دانست که باید در صورت نوشتن هر دو پراسس، مسائل مربوط به ناسازگاری دیتا را تحمل و مدیریت کند. بنابراین در سناریوهایی که پراسس‌ها نوشتن کمتری داشته باشند و بیشتر بخواهند از یک دیتای مشترک به عنوان خواننده استفاده کنند، این مکانیزم عملکرد بهتری خواهد داشت. در صورتیکه نیاز به نوشتن‌های متعدد روی دیتاهای مختلفی باشد، message passing عملکرد بهتری خواهد داشت.

سوال ٤:

تشریح وظیفه انواع زمان‌بندها:

- **کوتاه‌مدت:** پراسس بعدی‌ای که باید اجرا شود را انتخاب و منابع پردازشی را تخصیص می‌دهد.
- **بلندمدت:** پراسس‌هایی که باید در صف ready قرار گیرند را تعیین می‌کند.
- **میان‌مدت:** پراسس‌ها را در صورت لزوم از حافظه اصلی خارج، در حافظه دیسک ذخیره و در صورت رفع مشکلات دوباره بازگردانی می‌کند تا اجرا شوند.

A. زمان‌بند کوتاه‌مدت حتما اقدام و دقیقا وظیفه‌ای که در بالا تعریف شده را انجام خواهد داد. ممکن است زمان‌بند بلندمدت تغییراتی در صف ready انجام دهد.

B. درست مانند قسمت A

C. درست مانند A، با این تفاوت که پراسسی که درخواست I/O کرده است باید به صف I/O وارد شود که توسط زمان‌بند کوتاه‌مدت انجام می‌شود.

D. زمان‌بند میان‌مدت حتما وارد شده و اقدام تعریف شده در بالا را انجام می‌دهد. دو زمان‌بند دیگر هم احتمالا وارد شوند و وظایف تعریف شده را انجام دهند.

E. زمان‌بند کوتاه مدت یک پراسس را انتخاب خواهد کرد برای اجرا و زمان‌بند بلندمدت درخواست‌هایی که باید در صف ready قرار گیرند را تعیین می‌کند.

F. زمان‌بند کوتاه‌مدت سریعا آن را به حالت اجرا در می‌آورد.

G. زمان‌بند بلندمدت پراسس‌ها را به صف ready می‌برد و زمان‌بند کوتاه‌مدت یک پراسس را به حالت اجرا در می‌آورد.

سوال ٥:

الف)

- **()fork:** این سیستم‌کال برای ساختن یک پراسس جدید از یک پراسس والد استفاده می‌شود و ورودی‌ای ندارد. خروجی این سیستم‌کال ممکن است چند حالت داشته باشد:
  - در صورت موفقیت یک pid را به عنوان خروجی به والد برمی‌گرداند.
  - در صورت عدم موفقیت خروجی یک عدد منفی خواهد بود.

- خروجی صفر به پراسس فرزند ساخته شده.

- **exec():** خانواده توابع exec پراسس در حال اجرا را با پراسسی که مسیر فایل آن را به عنوان ورودی دریافت کرده جایگزین می‌کنند و پراسس جدید را اجرا می‌کنند.

ب)

سیستم‌کال fork() در هر بار اجرا یک کپی از فضای آدرس والد گرفته و فضای آدرس فرزند را به صورت اختصاصی در اختیار او قرار می‌دهد. به همین دلیل در صورت اجرای متوالی این تابع، فضای حافظه سیستم ممکن است که پر شود.

در خصوص exec() مشکل این است که این تابع پراسس قبلی را کاملا می‌بندد و با همان pid پراسس جدید را اجرا می‌کند. این موضوع می‌تواند برای زمان‌بندهای سیستم ایجاد اشکال کند چرا که درک درستی از زمان پایان این پراسس نخواهند داشت.

پ)

استفاده از سیستم‌کال ***vfork()***، فضای آدرس جدید و مجزایی را در اختیار پراسس فرزند قرار نمی‌دهد. به جای این کار، این تابع پراسس والد را به صورت موقت به حالت تعلیق در می‌آورد تا پراسس فرزند کارش به اتمام برسد چرا که هر دو والد و فرزند یک فضای آدرس را به اشتراک می‌گذارند.

مکانیزم *COW*، اگر یکی از این پراسس‌ها بخواهد به منابع مشترک حافظه دسترسی پیدا کند، روی صفحات حافظه علامت‌گذاری می‌کند تا هنگام نوشتن فقط روی کپی آن قسمت نوشته می‌شود بنابراین تغییرات روی آن کپی ایجاد می‌شود و روی پراسس دیگر تاثیر نمی‌گذارد.

سوال ٦: (با فرض *ppid*=١٠ و *pid*=١١)

الف)

- اگر ***fork()*** با موفقیت اجرا نشده باشد(*pid*<٠) :

pid: -١

- اگر ***fork()*** موفق باشد:
  - اگر بعد از سیستم‌کال، فرزند وارد شود:

ppid: ١٠
pid: ١١

  - اگر بعد از سیستم‌کال، والد وارد شود:

pid: ١١
ppid: ١٠

در این حالت به دلیل *cascading termination* بعد از اجرای کامل والد و *terminate* شدن آن، پراسس فرزند نیز *terminate* خواهد شد.

ب)

- اگر *fork()* با موفقیت اجرا نشده باشد(*pid<۰*) :

pid: -۱

- اگر *fork()* موفق باشد:
  - اگر بعد از سیستم‌کال، فرزند وارد شود:

ppid: ۱۰
pid: ۱۱
ppid: ۱۰

  - اگر بعد از سیستم‌کال، والد وارد شود:

pid: ۱۱
ppid: ۱۰
ppid: ۱۰

پ)

الف)

- اگر *fork()* با موفقیت اجرا نشده باشد:
  هیچکدام از *orphan* و *zombie* نخواهیم داشت.

- اگر *fork()* موفق باشد:
  بعد از اجرا و اتمام پراسس والد، فرزند یتیم می‌شود و به خاطر مکانیزم *cascading termination* پراسس فرزند هم کشته می‌شود.

ب)

- اگر *fork()* با موفقیت اجرا نشده باشد:
- اگر *fork()* موفق باشد:
  بعد از اجرا و اتمام پراسس والد، فرزند یتیم می‌شود و بلافاصله *init* سرپرستی فرزند را قبول میکند و قبل از اتمام پراسس روی آن *wait* می‌کند تا زامبی نشود. اگر در این فاصله *init* والد فرزند نشود و فرزند به حالت احتضار برود، تبدیل به زامبی می‌شود.

سوال ۷:

الف) خروجی کد:

۰
۱
۲
۵

در اجرای این کد، در هر قسمتی که *fork()* انجام می‌شود، یک پراسس فرزند ایجاد می‌شود. پس از آن والد برای پراسس ایجاد شده *wait()* می‌کند و فرزند از ادامه ی بلاک شرط *if* ادامه به اجرا می‌دهد. در انتهای اجرای هر پراسس، مقدار *sum*+۱ به عنوان *exit code* به عنوان خروجی داده می‌شود. به این ترتیب بعد از هر *wait()*، یعنی پس از اینکه هر فرزند با موفقیت اجرا شود، مقدار *exit code* آن به *sum* اضافه می‌شود.

ب)

به ازای هر خروجی (*sum*):

۰:

۱:

۲:

۵:

سوال ۸:

درخواست *shared memory* توسط پراسس‌ها باید به سیستم عامل ارسال شود. در صورت صلاحدید سیستم‌عامل فضای مشترک و آدرس این فضا را در اختیار پراسس‌ها خواهد گذاشت. تعیین اینکه به صورت همزمان نوشتن روی یک ناحیه خاص اتفاق نیوفتد، مسائل مربوط به مدیریت ناحیه بحرانی را مطرح می‌کند که راه‌حل‌های مختلفی برای آن ارائه شده است. در کل یک پراسس تنها در زمانی باید در ناحیه بحرانی خود باشد، که پراسس دیگری در ناحیه بحرانی خود نباشد.