دانشکده مهندسی کامپیوتر

دکتر رضا انتظاری ملکی

پاییز ۱۴۰۰

سیستم‌های عامل

پاسخنامه تمرین سری دوم

دانیال بازمانده – محمدعلی فراهت

سوال اول

دو تفاوت اصلی user-level threads و kernel-level threads را نام ببرید. هریک تحت چه شرایطی از دیگری بهتر است؟

پاسخ: سه مورد از مهم‌ترین تفاوت‌ها را می‌توان به صورت زیر بیان کرد:

۱. رشته‌های سطح کاربر (user-level threads) توسط kernel شناخته شده نیستند و بدون نیاز به kernel مدیریت می‌شوند؛ درصورتی که kernel از رشته‌های سطح kernel آگاه است و مدیریت آنها برعهده سیستم عامل است.
۲. رشته‌های سطح kernel معمولا نیازی به ارتباط با یک پردازه (process) ندارند، درصورتی که هر رشته‌ی سطح کاربر الزاما متعلق به یک پردازه است.
۳. در سیستم‌هایی که از نگاشت یک به چند (M:1) یا چند به چند (M:N) استفاده می‌کنند، رشته‌های سطح کاربر با استفاده از کتابخانه‌ی thread زمان‌بندی می‌شوند درصورتی که زمان‌بندی رشته‌های سطح kernel توسط خود kernel انجام می‌گیرد.

رشته‌های سطح kernel معمولا هزینه (cost) بیشتری نسبت به رشته‌های سطح کاربر دارند؛ زیرا توسط داده ساختارهای (data structures) خود kernel ذخیره و نمایش داده می‌شوند. بطور کلی، برای انجام کارهایی که نیاز به عملیات switch بیشتری بین رشته‌ها دارند، استفاده از رشته‌های سطح کاربر توصیه می‌شود. زیرا سرعت switch بین این رشته‌ها به دلیل یکجا بودن همه‌ی آنها در یک پردازه بیشتر است. در مقابل، برای انجام کارهایی که وقفه‌ها (interrupt) های زمانبری دارند (که عموما می‌تواند بخاطر عملیات I/O باشد) و هم‌چنین برای کارهایی که نیاز به تعداد رشته‌های بسیار بالا برای انجام دارند، استفاده از رشته‌های سطح kernel بهتر است.

سوال دوم

آیا یک راه حل multithreaded می تواند با استفاده از چند user-level thread در سیستم های چند پردازنده ای، عملکرد بهتری نسبت به سیستم های تک پردازنده ای ایجاد کند؟ توضیح دهید.

پاسخ: خیر. زیرا kernel از رشته‌های سطح کاربر تولید شده هیچ اطلاعی ندارد. بنابراین نمی‌تواند user-level thread ها را روی پردازنده‌های مختلف اجرا کند.

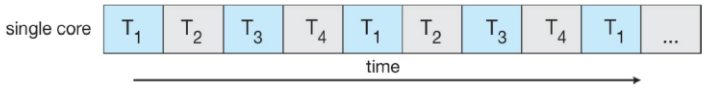
سوال سوم

پردازش موازی (Parallel) و همروند (Concurrent) را با رسم نمودار روند اجرا با یکدیگر مقایسه کنید.

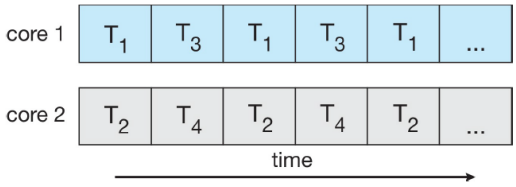
پاسخ: در پردازش موازی (Parallel) در هر لحظه می‌توانند چند تسک مختلف بطور همزمان درحال اجرا باشند؛ بطوری که تسک‌های موجود روی core ها تقسیم می‌شوند و هر تسک روی یک core جداگانه اجرا می‌شود. به همین علت، در سیستم‌های single core پردازش موازی معنایی ندارد.

در پردازش همروند (Concurrent) برخلاف پردازش موازی، روی یک core می‌تواند بیشتر از یک تسک اجرا شود اما در هر لحظه فقط یک تسک باید در حال اجرا باشد. می‌توانیم با استفاده از time sharing و switch های متوالی بین تسک‌ها روی یک core چندین تسک را در حال اجرا داشته باشیم.

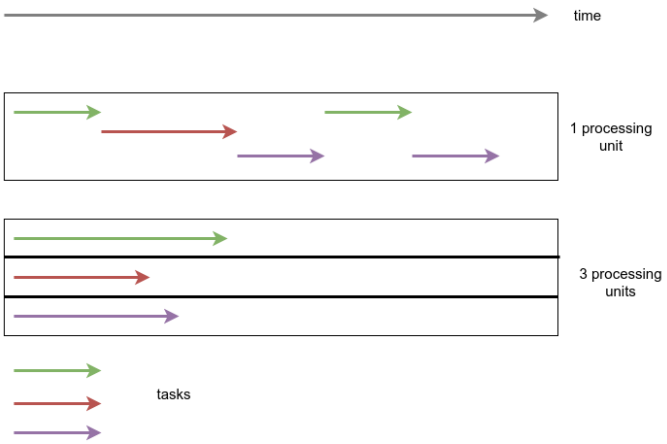
نمودارهای زیر می‌توانند درک بهتری از تعاریف بالا را ارائه دهند:



شکل ۱: پردازش همروند



شکل ۲: پردازش موازی



شکل ۳: مقایسه‌ی پردازش همروند (بالا) و پردازش موازی (پایین)

سوال چهارم

ذهن خلاق و بهینه‌دوست :) شما که بسیار به بازی‌های کامپیوتری علاقه دارد و از محدودیت‌های پردازشی موجود نیز باخبر است، به تازگی با مفهوم پردازش موازی آشنا شده و از شما می‌پرسد چرا به جای تعداد محدود هسته های محاسباتی موجود در پردازنده‌های امروزی، از تعداد بیشتری هسته (مثلا ۱۰۰ هسته!) استفاده نمی‌شود تا از سرعت‌های چند برابر بهره بگیریم؟

پاسخ شما چیست؟ (نرخ افزایش سرعت را برای ۸ ،۴ ،۲ و ۱۰۰ هسته محاسبه کنید.)

پاسخ: همانطور که می‌دانید در عمل، دستیابی به ۱۰۰ درصد از مولفه‌های موازی در یک برنامه حالت ایده‌آل است و امکان‌پذیر نیست و در هر برنامه‌ای مقدار قابل توجهی از برنامه باید بصورت serial اجرا شود. اگر درصد مولفه‌های serial یک برنامه را با S نشان دهیم و از رابطه‌ی قانون Amdahl کران بالای بهبود سرعت را به ازای core های مختلف محاسبه کنیم، داریم:

$$\text{speedup} \le \frac{1}{S + \frac{1-S}{N}}$$

$N = 2 \Rightarrow \text{speedup} \le \frac{2}{S+1}$ $\qquad$ $N = 4 \Rightarrow \text{speedup} \le \frac{4}{3S+1}$

$N = 8 \Rightarrow \text{speedup} \le \frac{8}{7S+1}$ $\qquad$ $N = 100 \Rightarrow \text{speedup} \le \frac{100}{99S+1}$

همانطور که مشاهده می‌شود، هرچه تعداد هسته‌های محاسباتی (core) افزایش می‌یابد، ضریب پشت S در مخرج کسر با شیب بسیار بیشتری نسبت به صورت کسر درحال افزایش است. این نتایج منجر به این می‌شود که از جایی به بعد به ازای S های غیرصفر، نمودار Speedup برحسب N به عدد خاصی همگرا شود. به عنوان مثال، اگر serial را ۲۵ و parallel را ۷۵ درصد درنظر بگیریم، داریم:

$N = 2 \Rightarrow \text{speedup} \le \frac{8}{5} = 1.6$ $\qquad$ $N = 4 \Rightarrow \text{speedup} \le \frac{16}{7} \simeq 2.3$

$N = 8 \Rightarrow \text{speedup} \le \frac{32}{11} \simeq 2.9$ $\qquad$ $N = 100 \Rightarrow \text{speedup} \le \frac{400}{103} \simeq 3.9$

همانطور که از نتایج مشاهده می‌شود، speedup تقریبا به عدد ۴ میل می‌کند.

پس می‌توانیم بگوییم افزایش بیش از حد تعداد هسته‌ها تاثیر آنچنانی روی افزایش سرعت ندارد و با افزایش هسته‌ها صرفا هزینه (cost) بیشتری را مصرف می‌کنیم.

سوال پنجم

یک سیستم multicore و یک برنامه multithreaded را که با استفاده از مدل many-to-many threading نوشته شده است، درنظر بگیرید. فرض کنید تعداد user-level thread ها در برنامه بیشتر از تعداد هسته های محاسباتی در سیستم باشد.

در هریک از حالت های زیر Performance را مورد بررسی قرار دهید.

(آ) تعداد kernel thread های اختصاص داده شده به برنامه از تعداد هسته های پردازشی کمتر باشد.

(ب) تعداد kernel thread های اختصاص داده شده به برنامه با تعداد هسته های پردازشی برابر باشد.

(ج) تعداد kernel thread های اختصاص داده شده به برنامه از تعداد هسته های پردازشی بیشتر باشد اما از تعداد user-level thread ها کمتر باشد.

پاسخ:

(آ) در این حالت بعضی از پردازنده‌ها بیکار می‌مانند، زیرا scheduler فقط kernel-level thread ها را به پردازنده نگاشت می‌کند، نه user-level thread ها را.

(ب) در این حالت این امکان وجود دارد که همه‌ی پردازنده‌ها هم‌زمان به‌کار گرفته شوند. اما اگر در یکی از پردازنده‌ها، kernel thread به هر دلیلی (page fault یا درحین انجام یک system call) مختل شود، آن پردازنده بیکار می‌ماند.

(ج) در این حالت همواره همه‌ی پردازنده‌ها به‌کار گرفته می‌شوند. چون حتی اگر یکی از kernel thread ها هم مختل شوند، یک kernel thread دیگر جایگزین آن می‌شود که این مورد، استفاده‌ی بهتری از سیستم‌های چند پردازنده‌ای را فراهم می‌کند.