سوال ۱)

الف)

kernel level thread	user level thread
پیاده سازی پیچیده توسط OS	پیاده سازی آسان توسط کاربر
زمان Context switch زیاد	زمان Context switch کم
آدرس حافظه مجزا	آدرس حافظه مشترک
نیاز به سخت افزار دارد.	بدون نیاز به سخت افزار می توان Context switch انجام داد
اگر یکی block شود، بقیه threadها به کار ادامه می دهند.	اگر یکی block شود تمام process متوقف خواهد شد.

ب) هر دو واحدهای اجرایی مستقل هستند. یکی از مهمترین تفاوت آنها این است که threadهای یک process با هم shared memory دارند اما در process از حافظه های مستقل استفاده می کنند.

می توان گفت که thread زیرمجموعه process هم می تواند باشد. اگر یک process، چند thread داشته باشد که به صورت موازی ران شوند.

یک thread زمان ساخته شدن منابع: stack، PC، Thread ID و مجموعه ای از رجیسترها را دارد. و code، file، data را به اشتراک می گذارد.

اما در ایجاد یک process یک PCB به آن اختصاص می دهیم که شامل memory map، open files list، variables است که این ساختار داده از threadها سنگین تر خواهد بود.

۱) ب) در الگوریتم های RR و FCFS گرسنگی رخ نمی‌دهد زیرا صفی وجود دارد که در هر صورت process ها
به ترتیب از آن خارج می‌شوند و در نهایت به همه‌ی آن‌ها رسیدگی می‌شود.

اما در SRTF ممکن است یک process به علت اینکه اینقدر زمان اجرایش طولانی باشد که هیچگاه نوبت
به آن نرسد و همواره process های کوتاه‌تری قبل از آن وارد صف شده و اجرا شوند.
برای حل این مشکل می‌توانیم از HRRN استفاده کنیم و aging و waiting time را در نظر بگیریم و
با توجه به آن اولویت process های قدیمی افزایش یافته و در نهایت
اجرا می‌شوند و دیگر دچار گرسنگی نخواهند شد.

۱) ت) همه ی switching در thread ها (در یک process) مخفی ها فضای یکسانی دارند و به همین

دلیل مقدار cache و memory address space تغیری نمی کند اما در process switching باید

PCB قبلی را ذخیره و PCB جدید را load کنیم، مقدار cache و TLB هم reset/flush می شوند زیرا

memory address space در آنها متفاوت است، هزینه ی این کارها زیاد است و در نتیجه زمان بیشتری می برد

Year:............ Month:............ Day:............

۱) ث) اگر همه ی فیلسوف ها همزمان قاشق سمت راست خود را بردارند، همه به حالت Hungry در

می آیند و همه منتظر قاشق سمت چپ می شوند و به دلیل نبودن آن نمی توانند غذاخوردن را شروع کنند و

در همین حالت به صورت waiting باقی خواهند ماند.

سوال ۲) در اینجا از fork برای ایجاد child process استفاده می‌کنیم. به این دلیل زمانی که
pid == 0 ← یعنی child process
pid > 0 مقدار value در آن ۲۰ می‌شود، اما چون پس از value ایجاد شده است با تغییر مقدار
مقدار value در parent تاثیری نمی‌گذارد همینطور که است.

اگر می‌خواستیم مقدار آن ۲۰ شود از vfork باید استفاده می‌کردیم) اما الان هر کدام از process ها روی variable های
مستقل عملیات انجام می‌دهند.

۳) x حداکثر ۱۰ خواهد بود (در صورتی که تردها با فاصله‌ی زمانی بتوانند کار خود را انجام دهند و تداخل نداشته باشند) برای این که می‌توانیم در نظر بگیریم که threadهای A و B به صورت یکی در میان به مقدار x اضافه می‌کنند.

همچنین با توجه به این که دستور اضافه کردن برای هر کدام حداقل ۵ بار انجام خواهد شد، کمترین مقدار صفر نمی‌تواند باشد. همچنین چون مقدار حلقه‌های تردها از یک بیشتر است، حداقل برابر $x = ۲$ می‌باشد. برای اثبات این

Year:............ Month:............ Day:............

ادامه‌ی سؤال ۳) موضوع آخرین ۱+ را در نظر بگیرید. قبل از آن حداقل یک بار این عمل توسط خود ترد یا ترد مقابل انجام شده است، پس مقدار $x = ۱$ است و با انجام آخرین مرحله $x = ۲$ می‌شود.

یک حالت برای $x = ۲$: ترد A و B همزمان حلقه‌ی اول را شروع کنند، ترد A به حلقه‌ی ۴ برسد و بلافاصله بعد از store آن، حلقه‌ی اول ترد B اتفاق بیفتد => $x = ۱$ پس حلقه‌ی دوم B و پنجم A همزمان شروع شوند و با انجام حلقه‌ی ۵ در B، حلقه‌ی ۵ A هم مقدار x را ۲ می‌کند (ذخیره کند).

توضیح: هر loop از load، inc و store تشکیل شده

A	B
load 1, inc 1, store 1,	load 1,
loop 2, loop 3, loop 4	inc 1,
load 5	store 1, $\Rightarrow x = 1$
inc 5	loop 2, loop 3 loop 4, loop 5
store 5 $\longrightarrow x = 2$	

همچنین تمام مقادیر بین ۲ تا ۱۰ هم برای x ممکن هستند (از تعداد loopهای بین دو ترد کم شود مقدار x بیشتر می‌شود)

Year:............ Month:............ Day:............

۴) طبق amdahl's law می‌دانیم:

$$\text{speed up} \le \frac{1}{s + \frac{(1-s)}{N}}$$

N: تعداد هسته، s: سریال برنامه‌ها

طبق این قانون از جایی به بعد با افزایش تعداد هسته دیگر شاهد افزایش سرعت نخواهیم بود، زیرا بخش پردازش بصورت سریال را محدود می‌کند هر چه این بخش سریال بزرگ‌تر باشد، مقدار زیاد شدن سرعت با زیاد کردن هسته‌ها کمتر می‌شود در صورتی که اگر نمودار آن را رسم کنیم به صورت حدی خواهد بود و مقدار آن کم و کمتر می‌شود

برای مثال با مقدار s های مختلف افزایش سرعت را بررسی می‌کنیم:

core \ s	۵٪	۲۰٪	۵۰٪
۲	$\frac{1}{0.05 + \frac{0.95}{2}} \simeq 1.9$	$\frac{1}{\frac{2}{10} + \frac{4}{10}} = 1.67$	$\frac{1}{\frac{1}{2} + \frac{1}{4}} = 1.33$
۴	$\frac{1}{0.05 + \frac{0.95}{4}} \simeq 3.48$	$\frac{1}{\frac{2}{10} + \frac{2}{10}} = 2.5$	$\frac{1}{\frac{1}{2} + \frac{1}{8}} = 1.6$
۸	$\frac{1}{0.05 + \frac{0.95}{8}} \simeq 5.92$	$\frac{1}{\frac{2}{10} + \frac{1}{10}} = 3.33$	$\frac{1}{\frac{1}{2} + \frac{1}{16}} = 1.78$
۱۰۰	$\frac{1}{0.05 + \frac{0.95}{100}} \simeq 16.8$	$\frac{1}{\frac{2}{10} + \frac{8}{10\times100}} = 4.8$	$\frac{1}{\frac{1}{2} + \frac{1}{200}} = 1.98$

⇐ افزایش تعداد هسته‌ها افزایش سرعت هم نرخ بیشتری پیدا می‌کند اما اگر تعداد هسته ثابت باشد با افزایش s این مقدار کمتر می‌شود.

Year:........ Month:........ Day:........

۵) الف)

FCFS:

P۱	P۲	P۳	P۴	P۵

۰ ۲ ۳ ۱۱ ۱۵ ۲۰

RR: q = 2

P۱	P۲	P۳	P۴	P۵	P۳	P۴	P۵	P۳	P۵	P۳

۰ ۲ ۳ ۵ ۷ ۹ ۱۱ ۱۳ ۱۵ ۱۷ ۱۸ ۲۰

SJF:

P۲	P۱	P۴	P۵	P۳

۰ ۱ ۳ ۷ ۱۲ ۲۰

non preemptive:

P۳	P۵	P۱	P۴	P۲

۰ ۸ ۱۳ ۱۵ ۱۹ ۲۰

ب)

turn around time:

	FCFS	RR	SJF	non preemptive
P۱	۲	۲	۳	۱۵
P۲	۳	۳	۱	۲۰
P۳	۱۱	۲۰	۲۰	۸
P۴	۱۵	۱۳	۷	۱۹
P۵	۲۰	۱۸	۱۲	۱۳

پ)

waiting time: turn around time - execution time

	FCFS	RR	SJF	non preemptive
P۱	۰	۰	۱	۱۳
P۲	۲	۲	۰	۱۹
P۳	۳	۱۲	۱۲	۰
P۴	۱۱	۹	۳	۱۵
P۵	۱۵	۱۳	۷	۸
sum	۳۱	۳۶	۲۳	۵۵

ت) در قسمت قبل مقدار sum برای هر الگوریتمی که بدست آمد میانگین زمان انتظار هم که ۵ برابر بود که در این الگوریتم SJF کمترین مقدار را دارد.

۶) الف) به منظور در اینجا از P = wait و V = signal است

با توجه به کد تولید کننده، اگر جای خالی n برای ایجاد وجود نداشته باشد، (یعنی empty = 0 شود) در همین سمت block می‌شود و پیش نمی‌رود.

ب) با توجه به wait در مصرف کننده برای سیگنال full می‌توان اطمینان پیدا کرد تا زمانی که مقدار آن از صفر بزرگتر نشود، اقدامی در جهت برداشتن بسته‌ها صورت نمی‌گیرد.

پ) خیر نمی‌توان از این موضوع اطمینان حاصل کرد. برای مثال اگر تولید کننده n دور اجرا شود و بعد با شروع حلقه‌ی for در مصرف کننده، تولید کننده هم می‌تواند حلقه‌ی for را (البته از هر کدام از مراحل به صورت یک به یک) اجرا کند؛ یعنی اگر تولید کننده وارد insert و مصرف کننده وارد remove شوند، این اتفاق خواهد افتاد. برای جلوگیری می‌توانیم یک سمافور باینری برای mutex تعریف کنیم.

ت) اگر سمافورهای full و empty به درستی انجام نشوند به بن‌بست یا deadlock برمی‌خوریم.

برای مثال اگر دو تا تولید کننده داشته باشیم که هر کدام به اندازه‌ی n/2 حداقل تولید کنند و همگام باشند، میزان جای خالی بزرگتر یا مساوی n است اما نمی‌توانند پیش بروند، همچنین در این حالت full = 0 است و مصرف کننده‌ای هم نمی‌تواند وارد عمل شود.