به نام خدا

حسین محصل ارجمندی ۹۴۳۱۷۰۳ پروژه اول سیستم عامل

//

سوال۱:

الف) الگوریتم دیفالت scheduler ، الگوریتم round robin است، با کوانتوم ۱. در هر tick، در حلقه for ای که کل جدول برنامه ها را بررسی می کند، ابتدا یک برنامه ای انتخاب می شود که RUNNABLE باشد و برای یک تیک، پردازنده در اختیار این برنامه قرار می گیرد. بعد از اتمام کار این برنامه، حلقه for بعد از این برنامه را بررسی می کند تا برنامه دیگری که RUNNABLE است را پیدا کند و cpu را به او بدهد. وقتی حلقه for مد نظر، که حلقه for داخلی است به اتمام رسید، از آنجا که بیرون این حلقه یک حلقه همواره صحیح قرار دارد، دوباره این حلقه for، از ابتدای جدول برنامه (یا در واقع همان صف برنامه ها) شروع کرده و دوباره برنامه RUNNABLE را انتخاب می کند.

ب) تابع allocproc، در جدول برنامه ها دنبال یک خانه خالی یا UNUSED می گردد، تا برنامه جدیدی که میخواهد به سیستم ما اضافه شود برای اجرا را initialize کند پارامتر هایش را، یعنیstate اش را EMBRYO می کند و به او یک process id (pid) اختصاص می دهد.

تابع wait، در کل جدول برنامه می گردد تا برنامه ای پیدا کند که فرزند برنامه فعلی باشد، اگر برنامه فعلی فرزندی نداشت ۱- بر می گرداند، در غیر این صورت انقدر صبر می کند تا فرزندش به اتمام برسد و state اش zombie بشود تا pid مربوط به فرزند را برگرداند.

تابع fork، یک پراسس جدید میسازد که فرزند پراسس فعلی در حال اجرا است. اگر موفق نشد، ۱- بر می گرداند و اگر موفق شد، به پدر pid فرزند را بر می گرداند و به فرزند، صفر بر می گرداند. State برنامه پدر و پشته آن را برای پروسه فرزند کپی می کند و حالت پروسه فرزند را RUNNALBE قرار می دهد.

ج) یک برنامه در حالت دیفالت، این attribute ها را دارد که در فایل proc.h قابل مشاهده است:

```
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)

};
```

د) تابع syscall، ابتدا شماره system call را از trapframe لود می کند، که شامل محتوای رجیستر a7 می شود. سپس، اندیس مورد نظر را در جدول system call ها جستجو می کند. برای اولین system call، a7 شامل مقدار SYS_exec است، و تابع syscall اندیس شماره SYS_exec جدول را فراخوانی می کند، که متناظر است با فراخوانی sys_exec. همچنین، تابع syscall مقدار بازگردانده شده از system call مد نظر را در داخل p->tf->a0 نگه می دارد. وقتی که فراخوانی system call به فضای برنامه کاربر بر می گردد، userret مقدار p->tf را داخل رجیستر های کامپیوتر لود می کنند و با استفاده از sret به فضای برنامه کاربر بر می گردند. بنابرین، وقتی exec از فضای برنامه کاربر بر می گردد، مقدار بازگردانده شده توسط system call handler را داخل a0 باز می گرداند. معمولا system call ها در شرایطی که موفقیت آمیز عمل نکنند، مقدار منفی بر می گردانند، و مقدار صفر یا مثبت بر می گردانند. اگر شماره system call معتبر نباشد، این تابع یک ارور چاپ می کند و ۱- بر می گرداند.

سوال ۲:

https://github.com/hm0ss/nst/commit/9cb4ca0a08dab1423fcc75109f401f666a050936

در این کامیت، proc.c عوض شده و تابع testgetchilds اضافه شده. از طرفی، برنامه testgetchilds.c اضافه شده که این تابع را تست می کند. سیستم کال testgetchilds، یک Int بر می گرداند که حاصل append شدن آی دی های فرزندان پروسه فعلی است، اگر یک رقمی باشد، به ابتدای آن ها یک ۰ اضافه می کند. سپس در تابع کاربر چاپ می شود.

سوال ۳:

۳,۱:

https://github.com/hm0ss/nst/commit/e1c784e7eb36300ba5361cc86cc3ada37056dfa5

در این کامیت، سیستم کال changepolicy تعریف شد که در صورت فراخوانی، مقدار defaultpolicy را از ۰ به ۱ یا از ۱ به صفر عوض می کند و بسته به مقدار defaultpolicy، scheduler میتواند تصمیم بگیرید کدام برنامه را انتخاب کند (تغییر scheduler جلوتر اعمل شده، فعلا فقط سیستم کال تعریف شده)

۳,۲:

https://github.com/hm0ss/nst/commit/14cc78e42f7c885f331dd6b43393d260e6da81da

در این کامیت، متغییر quantum به برنامه اضافه شده، ازین به بعد هر برنامه به تعداد quantum بار tick اجرا می شود. از طریق یک شمارنده به اسم counter که در هر بار اجرای cpu یک بار زیاد می شود تا به quantum برسد و سپس دوباره از صفر شروع می شود.

:۳,۳

https://github.com/hm0ss/nst/commit/c692e5d478d0b04cdb6a5891a92be3dbe2b21b81

در این کامت، هربار، scheduler مهم ترین اولویت (کمترین مقدار changeable priority) را برای اجرا انتخاب می کند. پس از یک tick، آن برنامه cpu را واگذار می کند و مقدار changeable priority اش به اضافه مقدار priority اش می شود. به صورت دیفالت، هر برنامه مقدار priority اش ۱۰ است و changeablepriority مقدار اولیه اش صفر است (همان cpriority) در این کامیت، سیستم کال Changepriotiy هنوز به درستی کار نمی کند و بعدا در کامیت مربوط به تست آن اصلاح می شود

:Extra implementation

https://github.com/hm0ss/nst/commit/a18eed1fe0bdf2c511b283c03dd7645a70b1b70d

در این کامیت هم متغیر های زمانی اضافه شده، هم تابع waitforchilds() و هم waitshowaverage(). در واقع تابع waitshowaverage() همان تابع waitforchildren() خواسته شده توسط سوال است، وقتی این تابع waitshowaverage() فراخوانی شود، پروسه فعلی انقدر منتظر می ماند تا تمام فرزندانش تمام شوند و اطلاعات زمانی خاسته شده توسط فرزندان چاپ می شود. برای چاپ شدن اطلاعات زمانی فرزندان، خطوط cprintf در تابع waitforchilds() از حالت کامنت خارج شوند.

:۳,٤

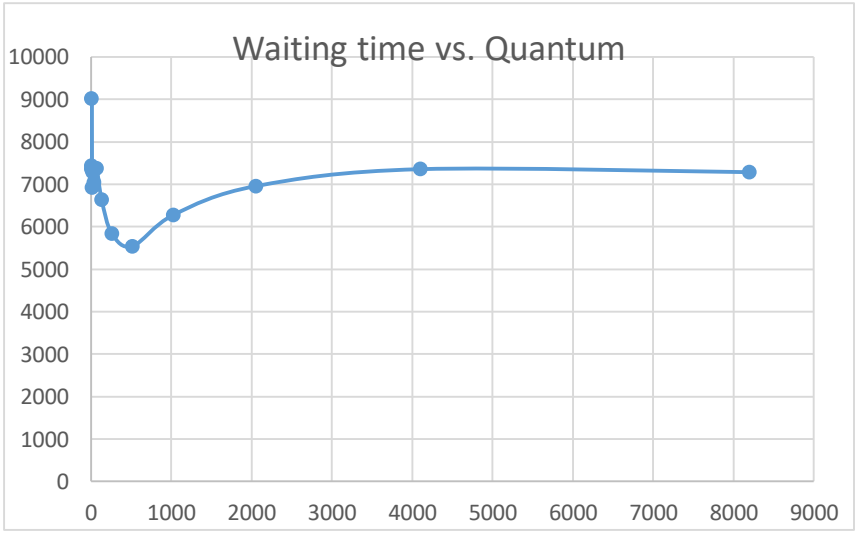
https://github.com/hm0ss/nst/commit/bb70dc7100be1e5008cecb38f4b60900b877040b

در این کامیت، multi level queue طراحی شده. به این ترتیب که سه صف وجود داریم، هر برنامه یک attribute به نام queue دارد که با سیستم کال changepolicy ایجاد شده که هنگام فراخوانی، صف برنامه فعلی را یکی به جلو شیفت می دهد. یعنی اگر صف اول باشد به صف دوم می رود، اگر صف دوم باشد به صف سوم می رود و اگر صف سوم باشد به صف اول بر می گردد.
همچنین، سیستم کال changeCurrentPriority افزوده شده که یک int میگیرد و الویت برنامه فعلی را به مقدار همان int فرار می دهد.
صف ها به صورت ذکر شده در سوال اند. صف شماره ۱ مربوط به صف اولویت، صف دوم مربوط به صف دیفالت، که همه برنامه های جدید به صورت پیش فرض در این صف هستند، و صف سوم که صف دیفالت است منتهی با مقدارquantum (که در ابتدا ۱ است).
در ضمن، فرض می شود که cpu به صورت نوبتی بین سه صف گردش می کند. در هر بار اجرای scheduler، با توجه به مقدار QQQ که مقدار شماره صف فعلی که برنامه آن باید اجرا شود را نشان میدهد. بعد از اینکه نوبت هر صف تمام شد، مقدار QQQ یکی بیشتر می شود و اگر ۳ بود دوباره یک می شود. توجه کنید که در حالت سوم چون کوانتوم داریم، QQQ تا زمانی که کوانتوم برنامه فعلی تمام نشود، همان ۳ میماند.
توجه شود که سیستم کال chanequantum، با هربار فراخوانی، کوانتوم فعلی را دوبرابر می کند. در واقع ما بررسی عملکرد را بر اساس توان های زوج کوانتوم انجام می دهیم.

:۳,۵,۱

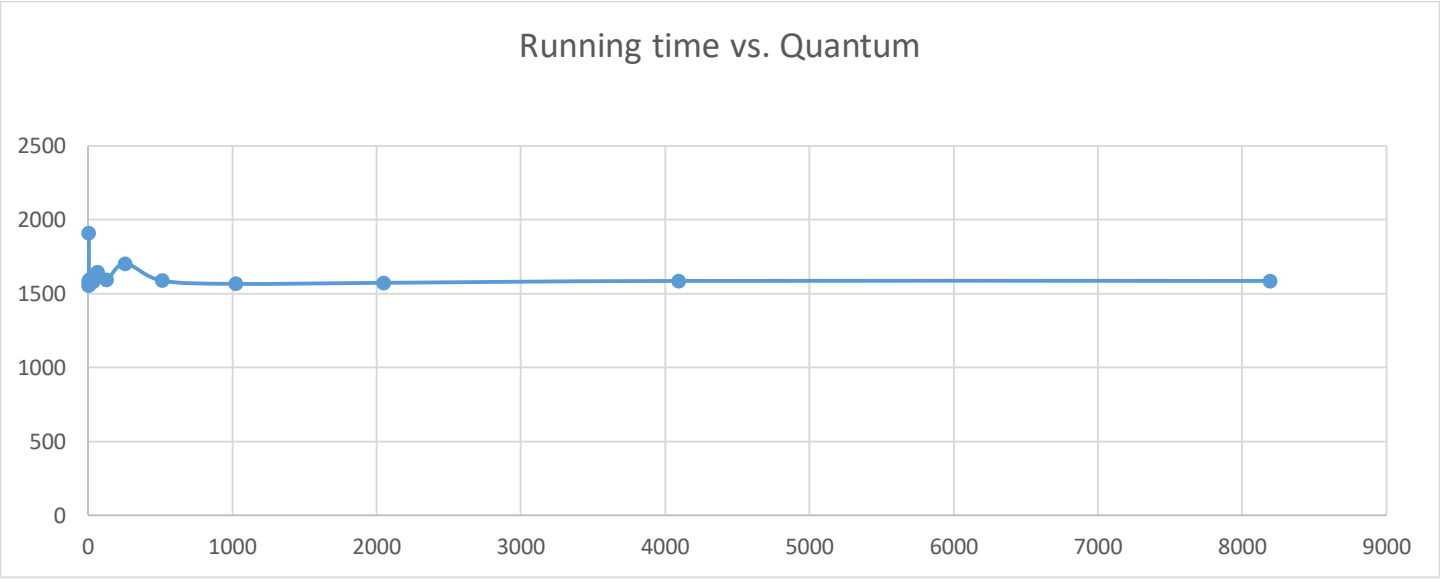
https://github.com/hm0ss/nst/commit/a18eed1fe0bdf2c511b283c03dd7645a70b1b70d

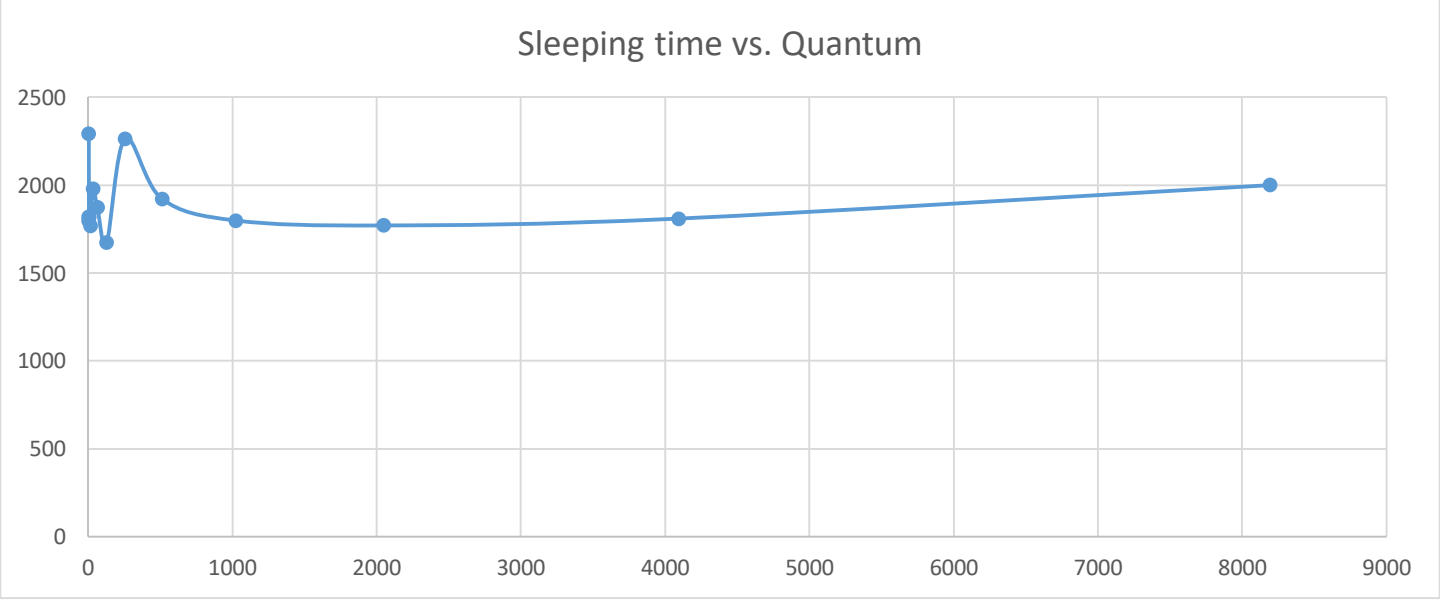
در کامیت بالا، مقادیر توان های دو کوانتوم امتحان شد (۱و۲و٤ و....). سپس در تعداد پروسس ثابت که کار های مشخصی انجام می دهند، مجموع زمان انتظار، مجموع زمان اجرا، مجموع زمان sleeping اندازه گیری شد و نمودار آن ها رسم گردید.
اکسل و نمودار ها ضمیمه شده اند.



در مورد تغییر عملکرد سیستم با تغییر کوانتوم، متوجه می شویم کهبه طور تقریبی، اول با دوبرابر شدن های متولی کوانتوم، از مقدار یه تا جایی، مقدار waiting time کل سیستم رو به کاهش است. اما از جایی به بعد، افزایش کوانتوم باعث افزایش wiaitingtime شده تا جایی که دیگر افزایش کوانتوم تاثیری روی waiting time ندارد.

تفسیر این پدیده این است که در کوانتوم های کوچک، سربار context switching زیاد است و باعث کندی برنامه ما می شود. با افزایش کوانتوم این مشکل تا حدی حل می شود، اما از جایی به بعد، به دلیل افزایش کوانتوم، دیگر برنامه ها همروند اجرا نمی شوند، یعنی کوانتوم انقدر زیاد شده که الگوریتم ما عملا شبیه FIFO شده، چون هر وقت نوبت برنامه ای برسد، کوانتومش آنقدر کافی است که این برنامه کارش را انجام دهد. در این شرایط هم عملا waiting time نسبا به حالت round robin بیشتر شده و البته از جایی به بعد چون با Fifo تفاوتی ندارد، تغییر کوانتوم دیگر اثری رو waiting time ندارد. مقدار Running time و sleeping time تقریبا مستقل از کوانتوم هستند.





به هر حال، با تغییر مقدار quantum در proc.c، یا با استفده از سیستم کال changequantum، با اجرای برنامه testquantum.c اطلاعات کلی عملکرد سیستم عامل قابل مشاهده است.

:۳,۵,۲

https://github.com/hm0ss/nst/commit/4c46f4aeefd8de877c5909df3e29b9d8d9097e43

در این کامنت، ابتدا سیستم کال ChangeCurrentPriority اصلاح شده، بدین صورت که وقتی فراخوانی شود، یک int دریافت می کند و اولویت پراسس فعلی را برابر مقدار int دریافت شده قرار می دهد. در برنامه testpriority.c، ابتدا چندین بار fork می کنیم، سپس به شاخه های مختلف اولویت های مختلف تخصیص می دهیم و مشاهده می کنیم که در کل، هرچقدر که اولویت عدد کمتری باشد (کمترین اولویت = ۱)، آن برنامه کمتر منتظر می ماند و زودتر پایان می یابد.

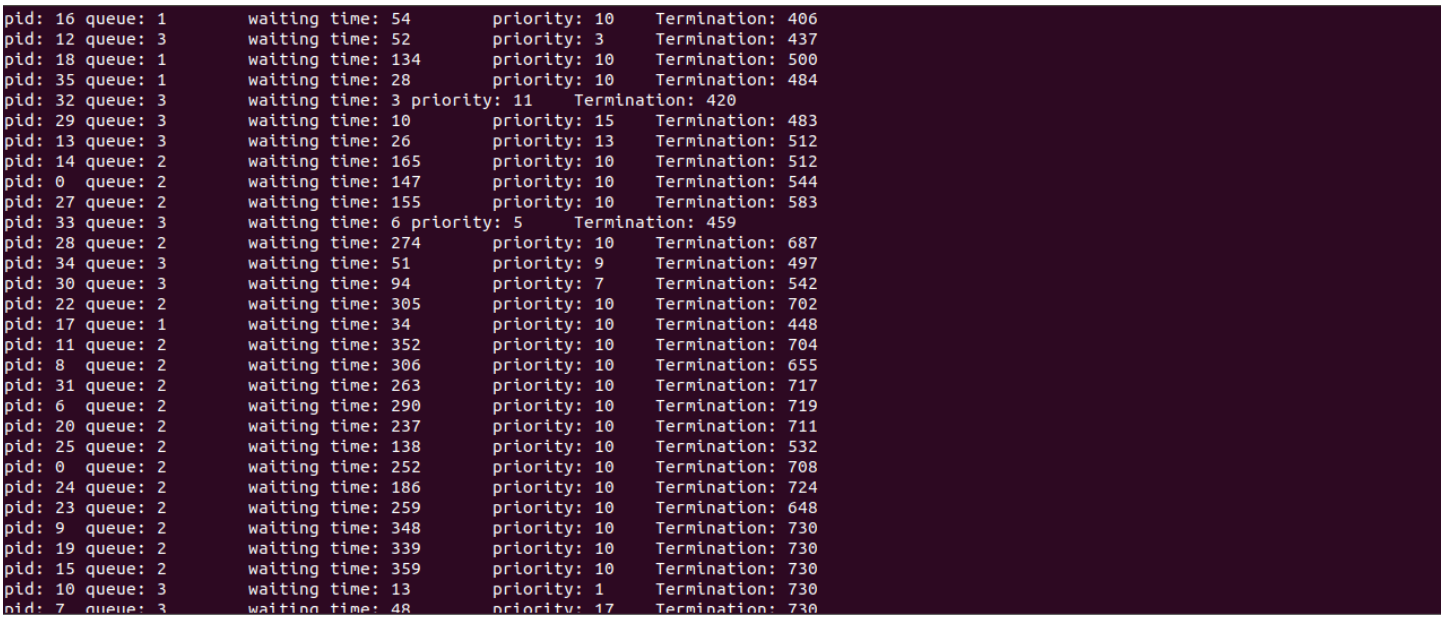
```
pid: 16 waiting time: 148       priority: 7     Termination: 567
pid: 19 waiting time: 180       priority: 10    Termination: 603
pid: 15 waiting time: 211       priority: 9     Termination: 632
pid: 27 waiting time: 1 priority: 1     Termination: 454
pid: 25 waiting time: 258       priority: 10    Termination: 710
pid: 26 waiting time: 271       priority: 10    Termination: 723
pid: 23 waiting time: 345       priority: 10    Termination: 802
pid: 10 waiting time: 95        priority: 10    Termination: 632
pid: 12 waiting time: 257       priority: 10    Termination: 711
pid: 17 waiting time: 143       priority: 5     Termination: 580
pid: 11 waiting time: 436       priority: 10    Termination: 856
pid: 35 waiting time: 410       priority: 10    Termination: 908
pid: 33 waiting time: 348       priority: 10    Termination: 908
pid: 31 waiting time: 360       priority: 10    Termination: 858
pid: 32 waiting time: 473       priority: 10    Termination: 982
pid: 30 waiting time: 456       priority: 10    Termination: 982
pid: 14 waiting time: 559       priority: 10    Termination: 984
pid: 34 waiting time: 490       priority: 10    Termination: 998
pid: 22 waiting time: 158       priority: 10    Termination: 998
pid: 0  waiting time: 97        priority: 3     Termination: 527
pid: 7  waiting time: 408       priority: 10    Termination: 856
pid: 13 waiting time: 587       priority: 10    Termination: 1012
pid: 8  waiting time: 450       priority: 10    Termination: 1012
pid: 20 waiting time: 555       priority: 10    Termination: 1014
pid: 28 waiting time: 545       priority: 10    Termination: 1028
pid: 21 waiting time: 581       priority: 10    Termination: 1035
pid: 9  waiting time: 479       priority: 10    Termination: 1035
pid: 5  waiting time: 578       priority: 10    Termination: 1035
pid: 29 waiting time: 546       priority: 10    Termination: 1036
pid: 24 waiting time: 231       priority: 10    Termination: 1036
```

:۳,۵,۳

https://github.com/hm0ss/nst/commit/bb70dc7100be1e5008cecb38f4b60900b877040b

در این کامیت، برنامه testmulti.c نوشته شده. ابتدا چندین بار fork می کنیم، سپس با استفاده از سیستم کال changepolicy، صف مربوط به برنامه فعلی را تغییر می دهیم. همه برنامه ها اول به صورت دیفالت در صم دوم هستند که همان صف دیفالت است. هر بار فراخوانی changepolicy معادل شیفت دادن صف به مقدار یک بار رو به جلو است، یعنی بار اول فراخوانی صف را از ۲ به ۳ تغییر می دهد و بار دوم فراخوانی، صف را از ۳ به ۱ تغییر می دهد. بدین طریق، صف بعضی از پراسس ها عوض می شود. از طرفی، سه بار changequantum فراخوانی می شود تا کوانتوم سیستم به ۸ تبدیل شود. لازم به ذکر است که کوانتوم فقط در صف سوم تاثیر دارد.

برای برنامه هایی که وراد صف دوم شده اند، با استفاده از سیستم کال changeCurrentPriority، مقدار اولویتشان عوض شده تا تاثیر اولویت را بر زمان پایان و مقدار انتظار ببینیم. در زیر، نتیایج را مشاهده می کنید:



همانطور که مشاهده می شود، در میان برنامه ها، برنامه هایی که در صف سوم بودند کمترین مقدار waiting time را داشتند. سپس، برنامه های صف اول مقدار waiting time کمتری داشتند نسبت به صف دوم، اما نسبت به صف اول مقدار waiting timeبیشتری داشتند. همچنین، مشاهده شد که در صف شماره ۲، هرچه اولویت برنامه پایین تر بود، مقدار waiting time برنامه کمتر شد.

نهایتا، برنامه های موجود در صف شماره ۲ یا همان صف دیفالت، بیشترین waiting time را داشتند.

همراه این فایل، آخرین نسخه از برنامه نیز آپلود شده، اما به جهت اینکه بعضا تغییرات رخ داده به صورت دنباله دار نبودند، و یک تابع ممکن است در کامیت های مختلف به شکل های مختلف باشد، بنابرین برای بررسی جواب هر سوال به کامیت مربوطه مراجعه شود.

همچنین، در صورت نیاز به هر گونه توضیح به من پیام بدین.

با تشکر.