Trabalho de Sistemas Operacionais

Tomás Rosário Rosemberg 14/0087567
Kevin Masida 13/0058866
Hichemm Khalyd 12/0012928

- Descrição das Ferramentas e linguagens usadas

A linguagem escolhida para a implentação do trabalho foi Python pois o integrante responsável pela a implentação já possuia uma experiencia com a linguagem. Por ela ser bastante usada para análise de dados e como o trabalho, em várias partes, poderia ser implementado usando lista, Python nos fornece varias facilidades para poder manipula-lás, tanto usando funções como map() e filter(), onde map, ao passaramos uma função e uma lista ela aplica a função a todos os elementos dessa lista e filter onde se é possível gerar uma nova lista a partir de uma existente filtrando elementos definidos por uma função. Outra facilidade que Python nos forneceu para a implementação do trabalho é o método append() das listas, que incorpora ao final da lista o elemnto passado, muito usado durante o trabalho e tambem a decomposição e inserção de elementos no meio da lista a partir de escolher posições da lista e iguala-los a uma nova lista.

-Descrição teórica e prática da solução dada

O trabalho consiste da implementação de um Sistema Operacional seguindo os requisitos explicitados na especificação entregue a nós pela professora. Nessas especificações dita os seguintes tópicos:

-Estruturação das filas de processo

Para a estruturação das filas de processos foi criada uma lista uma lista ordenada com os processos que foram lidos do arquivo de entrada, onde conforme o tempo passa esses processos serão adequadamente colocados nas filas corretas, sendo elas uma lista de processo de prioridade 0 (tempo real) e uma lista de processo de usuario, onde se guarda todos os processos de usuario que se encontram prontos para execução. Esses processos são alocados conforme sua prioriadade em três listas de prioridade um , dois ou três conforme o arquivo de entrada especifica.

A solução adotada para evitar starvation foi aumentar a prioridade de processos que não foram executados nenhuma vez nos ultimos 10 segundos, desta forma aumentando a probabilidade dele ser executado pois estará em uma lista de maior prioridade a partir deste momento. Por especificar que o quantum trabalhado deve ser de uma unidade de tempo, adotamos o sistema de *round robing* para que todos os processos possam ser executados , desta forma, ao ser executado, o processo volta para o final da fila de execuçã e abre oportunidade para outros processos serem executados também.

Caso o total da fila já possua 1000 processos e chegue um novo processo, esse processo será descartado visto que nas especificações não determina o que deve ser feito com ele e não há nela uma fila de processos a espera de poder se tornarem um processo pronto.

Estrutura de memória

Adotamos uma lista para representar a estrutura de memória desejada onde esta lista possui tamanho de 64 posições para processos de tempo real e 960 para processos de usuário, onde caso não haja processo na posição ela se encontra com None e caso tenha algo na posição ela se encontra um objeto do tipo processo que possui informação de PID, onde ele começa na memória e tamanho de memória que ele ocupa.

Estrutura de recursos disponíveis

Para a estruturação de recursos disponíveis foi criada uma classe que possui como atributo os recursos especificados onde cada um deles recebe uma lista de None como inicialização, caso algum recurso seja alocado por algum processo, o atributo da classe de gerenciamento de drivers se tornará True para que se tenha a informação de que ele esta alocado e não possa ser alocado por outro recurso.

Estrutura do Sistema de Arquivos

Para a estruturação do sistema de arquivos foi criado uma classe arquivo e uma classe de gerenciamento de arquivos que basicamente consiste em uma lista de arquivos, nesta classe fora implementados os métodos responsáveis pela alocação e exclusão de arquivos em disco baseando-se na manipulação de listas que o Python nos oferece como padrão.

Execução dos processos

Para se realizar como deve ser executado os processos seguimos os seguintes passos, após o processo estar alocado em sua fila correta, quando se começa um novo quantum, a lsita de processos de prioridade um sera varrida tentando alocar, em ordem, cada um dos processos da lista na memória disponivel para eles, após alocar todos os processo possíveis, nos direcionamos para as listas de usuário varrendo os processos a partir dos de menor prioriade (menor em valor da fila, estes são os processos com maior prioridade de serem executados) e checamos a memória disponível e se os driver externos estão disponiveis para os processos que os querem. Alocando todos os processos possíveis a partir desta lógica, executamos uma instrução de todos eles e os processos de tempo real retornam para suas filas adequedas e são retirados de memória e liberam os drivers que tinha acesso. Como os processos de tempo real são implementados como FIFO eles não saem da memória após a passagem do quantum, apenas quando terminam sua execução.

Ao recebermos as informações na entrada sabemos quanto tempo de processamento cada processo deverá receber, desta forma se ele tiver mais instruções para executar do que tempo de processamento dedicado a ele, ele não executará suas ultimas intruções, somente as que der tempo. Porém, se ele tiver menos instruções para executar do que tempo de memória dedicado a ele, ele continuará possuindo a CPU durante o tempo dedicado a ele, mesmo sem instruções.

Principais dificuldades durante a implentação

As maiores dificuldades durante a implentação foram adequar as saidas a forma desejada pela professora e a modelagem do trabalho. Antes de começar o trabalho foi feito um estudo a risca das especificações e foi elaborado um papel e um arquivo de texto especificando como algumas partes que pareciam mais problematicas deveriam ser implementadas, desta forma o trabalho demorou mais do que o normal para ser inicializado, porém isso evitou muitos problemas durante a implementação. A decisão de separar cada grande módulo do trabalho em um arquivo diferente facilitou a implementação do trabalho pois era mais fácil de localizar os erros e estruturar onde cada função deveria ser implementada no trabalho.

Papel de cada integrante do grupo na realização do trabalho

O integrante Tomás Rosário Rosemberg foi o integrante responsável pela modelagem e estudo de como o trabalho seria efetuado, implementação de toda a lógica do programa e também pelo relatório do trabalho, logo o trabalho foi feito totalmente por ele.

Bibliografia

Slides da matéria de Sistemas Operacionais da professora Aletéia

Site StackOverflow para tirar duvidas a certa da implementação lógica de alguns módulos do trabalho.

O trabalho se encontra no repositório do aluno Tomás Rosário Rosemberg no GitHub pois ele foi usado para versionamento de código e implementação em grupo do trabalho.
https://github.com/trosemberg/SO_UNB

Arquivo Main.py

```
import sys
import processo as gproc
import memoria as gmem
import recursos as ges
import arquivos as garq
from input_output import gera_processos,gera_arquivos, Output

"""
Os imports estao sendo feitos como g+primeira letra do import pois e gerenciador de...
es = entrada e saida
"""

def start():
"""
variavel memoria e responsavel pelo gerenciamento de memoria,
variavel drivers e responsavel pelo gerenciamento dos drivers
variavel disco e responsavel pelo gerenciamento dos arquivos em disco
variavel g_proc e responsavel pelo gerenciamento de processos
variavel saida e responsavel pro printar na tela as saidas do programa
"""
memoria = gmem.G_Memoria()
drivers = ges.G_Drivers()
saida = Output()
processos = []
fileProc = "processes.txt"
fileFiles = "files.txt"
#checa se teve como entrada o nome dos arquivos a serem lidos, se sim seta pra eles a leitura
if (len(sys.argv) == 3):
fileProc = sys.argv[1]
fileFiles = sys.argv[2]
#abre o arquivo a ser lido e gera um vetor de processos.
processos = gera_processos(fileProc)
[vetor,instructions] = gera_arquivos(fileFiles)
disco = garq.G_Arquivos(vetor)
for processo in processos:
instructions = processo.set_inst(instructions)
#cria o gerenciador de processos
g_proc = gproc.G_Processos(processos)
tempo = 0
#enquanto houver processos a serem para entrarem nas filas ou processos de usuarios ou processos
#de tempo real de tempo real sendo executados (nao precisa testar se tem processo
# de usuario sendo executado, visto que nao tem nenhum sendo no momento da checagem)
while(g_proc.fora_filas or g_proc.usuario or g_proc.fila_p0 or g_proc.exec_real):
# checa se chegou processo no tempo e encaminha a fila certa
g_proc.org_filas(tempo)
# altera prioridade de processos que estao esperando a muito tempo sem ser executado
g_proc.altera_prioridade(tempo)
#varre fila de tempo real e se for possivel alocar processo, tira da fila de tempo real
# coloca na fila de execucao
for processo in g_proc.fila_p0:
if memoria.aloca_processo(processo):
g_proc.exec_real.append(processo)
```

```
g_proc.fila_p0 = filter(lambda x: x!= processo,g_proc.fila_p0)
saida.print_dispatcher(processo)
#varre fila de prioridade 1 e se for possivel alocar processo, tira da fila
# e coloca na fila de execucao de processos de usuario
for processo in g_proc.fila_p1:
if(drivers.get_drivers(processo)):
if memoria.aloca_processo(processo):
g_proc.exec_user.append(processo)
g_proc.fila_p1 = filter(lambda x: x!= processo,g_proc.fila_p1)
g_proc.usuario = filter(lambda x: x!= processo,g_proc.usuario)
saida.print_dispatcher(processo)
#varre fila de prioridade 2 e se for possivel alocar processo, tira da fila
# e coloca na fila de execucao de processos de usuario
for processo in g_proc.fila_p2:
if(drivers.get_drivers(processo)):
if memoria.aloca_processo(processo):
g_proc.exec_user.append(processo)
g_proc.fila_p2 = filter(lambda x: x!= processo,g_proc.fila_p2)
g_proc.usuario = filter(lambda x: x!= processo,g_proc.usuario)
saida.print_dispatcher(processo)
#varre fila de prioridade 3 e se for possivel alocar processo, tira da fila
# e coloca na fila de execucao de processos de usuario
for processo in g_proc.fila_p3:
if(drivers.get_drivers(processo)):
if memoria.aloca_processo(processo):
g_proc.exec_user.append(processo)
g_proc.fila_p3 = filter(lambda x: x!= processo,g_proc.fila_p3)
g_proc.usuario = filter(lambda x: x!= processo,g_proc.usuario)
saida.print_dispatcher(processo)
#executa as instrucoes comeca aqui
result = ""
for processo in g_proc.exec_real:
processo.execucao +=1
#so executa se tiver instrucoes a serem executadas
if (processo.execucao<=len(processo.instruc)):
saida.print_instructions(processo)
result = disco.executa(processo)
saida.log_instrucoes.append(result)
#retira processo se ja executou as vezes necessarias
if processo.execucao>=processo.t_proc:
memoria.retira_processo(processo)
result = ""
for processo in g_proc.exec_user:
processo.execucao +=1
#executa as instrucoes equivalente ao tempo de processamento
if (processo.execucao<=len(processo.instruc)):
saida.print_instructions(processo)
result = disco.executa(processo)
saida.log_instrucoes.append(result)

#fim da execucao
#devolve ao final da fila os processos de usuario que foram executados menos vezes do
#que o tempo de processamento do processo
g_proc.limpa_fila_exec_real()
g_proc.proc_user_fila()
#limpa a memoria de processos de usuario
memoria.limpa_memoria_usuario()
#libera os drivers que foram alocados
drivers.free_drives()
#vai para o proximo tempo
tempo+=1
#printa o log de processos
saida.print_log()
#printa o mapa de disco
saida.print_disco(disco)
print("instrucao sem processo para ser executada {}".format(instructions))
if __name__ == '__main__':
start()
```

Aquivo Arquivos.py

```
"""
Classe responsavel por gerenciar os arquivos em disco
"""

class G_Arquivos:
#init e a inicializacao do gerenciador de arquivos, onde ja inicializa com as infos passadas
```

```
def __init__(self,vetor):
self.blocos = []
self.blocos_total = int(vetor[0])
for _ in range(0,self.blocos_total):
self.blocos.append(Arquivos())
vetor.pop(0)
for element in vetor:
temp = element.split(',')
for i in range(0,len(self.blocos)):
pos_ini = int(temp[1])
pos_fim = pos_ini + int(temp[2])-1
if ((i>=pos_ini )and (i<=pos_fim)):
self.blocos[i].start = pos_ini
self.blocos[i].name = temp[0]
self.blocos[i].size = pos_fim-pos_ini+1
#classe responsavel por retornar a string q sera printada caso chame print(G_Arquivos)
def __str__(self):
string = "MAPA DOS BLOCOS DO DISCO:\n\t"
for bloco in self.blocos:
string += str(bloco)
string += "|"
return string
"""
Funcao responsavel por executar as intrucoes do processo com base na
execucao atual e posicao do vetor de instrucoes do processo.
Retorna uma string com o log para ser impresso na tela depois.
"""
def executa(self,processo):
string = ""
livre = 0
if processo.execucao <=len(processo.instruc):
operacao = processo.instruc[processo.execucao -1][0]
nome_arq = processo.instruc[processo.execucao -1][1]
#op 0 = criar op 1 = deletar
if operacao == 0:
tamanho_arq = processo.instruc[processo.execucao -1][2]
for i in range(0,self.blocos_total):
if(self.blocos[i].name==0):
livre +=1
if livre == tamanho_arq:
pos = i-tamanho_arq+1
self.blocos[pos:i+1] = livre * [Arquivos(nome_arq,pos,tamanho_arq,processo.PID)]
string = "Sucesso!\n"
string += "O processo {} criou o arquivo {}".format(processo.PID,nome_arq)
string += "(do bloco {} a {}).".format(pos,pos+tamanho_arq-1)
break
else:
livre = 0
string = "Falha!\n"
string += "O processo {} ".format(processo.PID)
string += "nao pode criar o arquivo {} (falta de espaco).".format(nome_arq)
return string
elif operacao == 1:
for i in range(0,self.blocos_total):
if(self.blocos[i].name == nome_arq):
if((processo.PID == self.blocos[i].creator) or processo.prioridade == 0):
size = self.blocos[i].size
self.blocos[i:i+size] = size*[Arquivos()]
string = "Sucesso!\n"
string += "O processo {} deletou o arquivo {}.".format(processo.PID,nome_arq)
break
else:
string = "Falha!\n"
string+= "O processo {} nao pode deletar o arquivo {}".format(processo.PID,nome_arq)
else:
string = "Falha!\n"
string += "Nao existe o arquivo {}".format(nome_arq)
return string
else:
return "Operacao nao existente"
else:
return processo.PID

"""
Classe que representa um arquivo que sera armazenado em disco (G_Arquivos)
"""
class Arquivos:
def __init__(self,nome = 0,start = 0,size = 0, creator = None ):
self.name = nome
```

```
self.start = start
self.size = size
self.creator = creator

def __str__(self):
return ("| {} ".format(self.name))
```

Arquivo Input_Output.py

```
import processo as gproc
"""
Arquivo responsavel pelas funcoes que farao a leitura dos arquivos de entrada
e pela saida do programa.
"""

"""
Gera os processos a partir do primeiro arquivo
"""
def gera_processos(fileProc):
id = 0
processos= []
with open(fileProc,"r") as input_proc:
for linha in input_proc:
processos.append(gproc.Processos([int(x) for x in linha.split(',')]))
#arruma o vetor de processo de menor tempo a maior tempo de inicializacao
processos.sort(key = lambda x:x.t_init,reverse = False)
#arruma o id de cada processo com base no tempo de inicializacao.
for processo in processos:
processo.PID = id
id+=1
return processos

"""
Responsavel por ler o segundo arquivo e retornar um vetor que sera usado para a criacao
da memoria de arquivos e as instrucoes para serem colocadas nos processos
"""
def gera_arquivos(fileFiles):
vetor = []
with open (fileFiles,"r") as input_file:
temporario = input_file.read().splitlines()
vetor.append(int( temporario[0]))
i = int(temporario[1])
for it in range(2, i+2):
vetor.append(temporario[it])
instructions = []
for it in range(i+2,len(temporario)):
instructions.append(temporario[it])
return [vetor,instructions]

"""
classe responsavel por dar as saidas do programa
"""

class Output:
def __init__(self):
self.log_instrucoes = []
"""
Imprime o mapa do disco
"""
def print_disco(self,disco):
print(disco)
"""
imprime a mensagem do dispatcher do referido processo
"""
def print_dispatcher(self,processo):
print('dispatcher =>')
print('\tPID:\t\t {}'.format(processo.PID))
print('\toffset:\t\t {}'.format(processo.pos))
print('\tblocks:\t\t {}'.format(processo.mem_bloc))
print('\tpriority:\t {}'.format(processo.prioridade))
print('\ttime:\t\t {}'.format(processo.t_proc))
print('\tprinters:\t {}'.format(bool(processo.impressora)))
print('\tscanner:\t {}'.format(bool(processo.scanner)))
print('\tmodem:\t\t {}'.format(bool(processo.modem)))
print('\tdrives:\t\t {}'.format(bool(processo.sata)))
"""
imprime o log das operacoes executadas
"""
```

```
def print_log(self):
operacao = 0
print('Sistema de Arquivos =>')
for log in self.log_instrucoes:
operacao +=1
print("Operacao {}=>".format(operacao))
print(log)
"""
imprime se o processo comecou, a instrucao executada e se acabou o processo
"""
def print_instructions(self,processo):
if processo.execucao == 1:
print("P{} STARTED".format(processo.PID))
print("P{} instruction {}".format(processo.PID,processo.execucao))
if processo.execucao == processo.t_proc:
print("P{} return SIGINT".format(processo.PID))
```

Arquivo Memoria.py

```
REAL_SIZE = 64
USER_SIZE = 960
"""
Unidade responsavel pelo gerenciamento da memoria
"""
class G_Memoria:
def __init__(self):
self.memoria = [None for _ in range(0, REAL_SIZE+USER_SIZE)]
"""
aloca o processo na parte correta da memoria e retorna True se foi possivel alocar e
False se nao foi possivel alocar
"""
def aloca_processo(self,processo):
livre = 0
if (processo.prioridade == 0):
inicio = 0
fim = REAL_SIZE
else:
inicio = REAL_SIZE
fim = REAL_SIZE+USER_SIZE
for i in range(inicio,fim):
if(self.memoria[i] == None):
livre +=1
if(livre == processo.mem_bloc):
processo.pos = i- processo.mem_bloc + 1
self.memoria[processo.pos:processo.pos + livre] = livre * [processo.PID]
return True
else:
livre = 0
return False
"""
Funcao responsavel por retirar o processo que
ja tiver terminado
"""
def retira_processo(self,processo):
self.memoria[processo.pos:processo.pos + processo.mem_bloc] = [None]*processo.mem_bloc

"""
Funcao responsavel por limpar a memoria de usuario
"""
def limpa_memoria_usuario(self):
self.memoria[REAL_SIZE:] = USER_SIZE * [None]
```

Arquivo Processo.py

```
import input_output as output
TAM_MAX = 1000

"""
Responsavel pela parte de Processos
"""
class Processos:
def __init__(self, processo):
self.t_init = processo[0]
self.prioridade = processo[1]
self.t_proc = processo[2]
self.mem_bloc = processo[3]
```

```
self.impressora = processo[4]
self.scanner = processo[5]
self.modem = processo[6]
self.sata = processo[7]
self.PID = None
self.pos = None
self.execucao = 0
self.prio_changed = False
self.instruc = []
def __str__(self,flag = "normal"):
if (flag=="normal"):
return ("t_init= {}, prio ={}," \
"id= {} inst= {}".format(self.t_init, self.prioridade,self.PID,self.instruc ))
elif (flag == "fila"):
return("|id:{}".format(self.PID))
"""
Responsavel por receber o vetor de instrucoes num formato de string
e associar corretamente ao processo do qual ele vem
o formato do vetor instrucoes ficou como exemplo
[[0,'B',2],[0,'D',3]]
"""
def set_inst(self,instructions):
rest = instructions
for inst in instructions:
real_inst = inst.split(',')
if (self.PID == int(real_inst[0])):
rest = filter(lambda x : x != inst, rest)
if (int(real_inst[1])== 0):
self.instruc.append([int(real_inst[1]),real_inst[2][1],int(real_inst[3])])
else:
self.instruc.append([int(real_inst[1]),real_inst[2][1]])
return (rest)

"""
Responsavel pelo gerenciamento de processos
"""

class G_Processos:
def __init__(self, processos):
self.fila_p0 = []
self.fila_p1 = []
self.fila_p2 = []
self.fila_p3 = []
self.usuario = []
self.fora_filas = processos
self.exec_user = []
self.exec_real = []
self.output = []

def __str__(self):
string = ""
string+= "p0:"
for processo in self.fila_p0:
string += str(processo.__str__("fila"))
string+="\n"
string+= "p1:"
for processo in self.fila_p1:
string += str(processo.__str__("fila"))
string+="\n"
string+= "p2:"
for processo in self.fila_p2:
string += str(processo.__str__("fila"))
string+="\n"
string+= "p3:"
for processo in self.fila_p3:
string += str(processo.__str__("fila"))
string+="\n"
string+= "fora:"
for processo in self.fora_filas:
string += str(processo.__str__("fila"))
string+="\n"
return string
"""
Responsavel por organizar as filas, setando para os lugares certos os processos
"""
def org_filas(self,tempo):
for processo in self.fora_filas:
if((len(self.fila_p0)+len(self.usuario) >=1000)and (processo.t_init<= tempo)):
self.fora_filas = filter(lambda x : x!= processo, self.fora_filas)
```

```
break
if ((processo.t_init<= tempo)):
if(processo.prioridade == 0):
self.fila_p0.append(processo)
self.fora_filas = filter(lambda x : x != processo,self.fora_filas)
if(processo.prioridade == 1):
self.fila_p1.append(processo)
self.usuario.append(processo)
self.fora_filas = filter(lambda x : x != processo,self.fora_filas)
if(processo.prioridade == 2):
self.fila_p2.append(processo)
self.usuario.append(processo)
self.fora_filas = filter(lambda x : x != processo,self.fora_filas)
if(processo.prioridade == 3):
self.fila_p3.append(processo)
self.usuario.append(processo)
self.fora_filas = filter(lambda x : x != processo,self.fora_filas)
"""
Altera a prioridade dos processos que estiverem a 10 unidades de tempo esperando
sem nunca terem sidos executados
"""
def altera_prioridade(self,tempo):
for processo in self.fila_p2:
if((processo.execucao == 0) and (processo.t_init + 9 < tempo) and (not processo.prio_changed)):
self.fila_p2 = filter(lambda x : x != processo, self.fila_p2)
processo.prio_changed = True
self.fila_p1.append(processo)
elif((processo.execucao == 0) and (processo.t_init + 19 < tempo) and (processo.prio_changed)):
self.fila_p2 = filter(lambda x : x != processo, self.fila_p2)
self.fila_p1.append(processo)
for processo in self.fila_p3:
if((processo.execucao == 0) and (processo.t_init + 9 < tempo) and (not processo.prio_changed)):
self.fila_p3 = filter(lambda x : x != processo, self.fila_p3)
processo.prio_changed = True
self.fila_p2.append(processo)
"""
funcao responsavel por, apos a execucao,devolver ao final da fila os
processos que nao terminaram execucao
"""
def proc_user_fila(self):
for processo in self.exec_user:
if processo.execucao >= processo.t_proc:
self.exec_user = filter(lambda x : x != processo,self.exec_user)
self.output.append(processo)
else:
if(processo.prioridade == 1):
self.fila_p1.append(processo)
self.usuario.append(processo)
self.exec_user = filter(lambda x : x != processo,self.exec_user)
if(processo.prioridade == 2):
self.fila_p2.append(processo)
self.usuario.append(processo)
self.exec_user = filter(lambda x : x != processo,self.exec_user)
if(processo.prioridade == 3):
self.fila_p3.append(processo)
self.usuario.append(processo)
self.exec_user = filter(lambda x : x != processo,self.exec_user)
"""
Tira os processos concluidos da fila de execuacao dos reais
"""
def limpa_fila_exec_real(self):
for processo in self.exec_real:
if (processo.execucao >= processo.t_proc):
self.exec_real = filter(lambda x : x != processo,self.exec_real)
self.output.append(processo)
```

Arquivo Recurso.py

```
"""
Classe responsavel pelo gerenciamento dos drivers a serem utilizados
"""
class G_Drivers:
def __init__(self):
self.scanner = [None]
self.impressora = [None]
self.modem = [None]
self.sata = [None,None]
```

```
"""
Funcao responsavel por alocar o driver pro processo e retornar True
caso algo tenha sido alocado que foi pedido, True se nada foi pedido pra ser
alocado e False caso nao tenha sido possivel alocar o requisitado
"""
def get_drivers(self,processo):
is_driver_free = True
if (processo.scanner == 1) and (self.scanner[0] is not None):
is_driver_free = False
if (processo.impressora == 1) and (self.impressora[0] is not None):
is_driver_free = False
if (processo.modem == 1) and (self.modem[0] is not None):
is_driver_free = False
if (processo.sata == 1) and (self.sata[0] is not None):
is_driver_free = False
if (processo.sata == 2) and (self.sata[1] is not None):
is_driver_free = False
if is_driver_free:
if(processo.scanner == 1):
self.scanner[0] = True
if(processo.impressora == 1):
self.impressora[0] = True
if(processo.modem == 1):
self.modem[0] = True
if(processo.sata == 1):
self.sata[0] = True
if(processo.sata == 2):
self.sata[1] = True
return is_driver_free

"""
Funcao responsavel por liberar todos os drivers , deve ser usada apos o termino do quantum
libera todos os drivers independente do processo pois os processos de tempo_real
nao podem acessar driver e os de usuario duram um quantum sendo usado, logo
eles alocam, executam as instrucoes e desalocam.
"""
def free_drives(self):
self.scanner = [None]
self.impressora = [None]
self.modem = [None]
self.sata = [None,None]
```