Table of Contents

Operating system မိတ်ဆက်

OS လို့ခေါ်တဲ့ operating system ဆိုတာ ကျနော်တို့နဲ့ စိမ်းတဲ့ အရာတစ်ခုတော့ မဟုတ်ပါဘူး။ ကျနော်တို့နေ့တိုင်းလိုလို ထိတွေ့နေရတဲ့ အရာတစ်ခုဖြစ်ပါတယ်။ smartphone မှာဆိုလည်း android, ios . computer မှာဆိုရင်လည်း window, linux , mac စသည်ဖြင့် operating system တော်တော်များများနဲ့ ကျနော်တို့နေ့စဉ်ထိတွေ့နေရတာပဲဖြစ်ပါတယ်။

ကျနော်တို့နေ့စဉ် အသုံးပြုနေတဲ့ applications တွေ run နိုင်ဖို့အတွက် hardware resource တွေလိုအပ်တယ်။ Hardware ရှိယုံနဲ့လည်း တန်း run လို့မရသေးပါဘူး။ application programs တွေအဆင်ပြေပြေ run နိုင်ဖို့အတွက် Operating System လိုအပ်ပါတယ်။ တစ်နည်းအားဖြင့် ကျနော်တို့အသုံးပြုနေတဲ့ applications တွေနဲ့ hardware တွေကြားမှာ OS

ရှိပါတယ်။ ဒီထက်ပိုပြီးရှင်းအောင်ပြောရရင် OS က hardware တွေကို manage လုပ်ပါတယ်။ ကျနော်တို့ အသုံးပြုနေတဲ့ applications တွေကို OS ပေါ်မှာ တင် run ပါတယ်။ ဥပမာ ကျနော် အခု Microsoft word သုံးနေတယ်။ Microsoft word ကနေတန်းပြီးတော့ directly hardware resource တွေပေါ် မကိုင်တွယ်ပါဘူး။ ကျနော်အသုံးပြုနေတဲ့ OS ကသာ word နဲ့ hardware resource တွေကြားထဲမှာ interface တစ်ခုအနေနဲ့ ရှိနေပြီး handling လုပ်ပေးနေတာပါ။

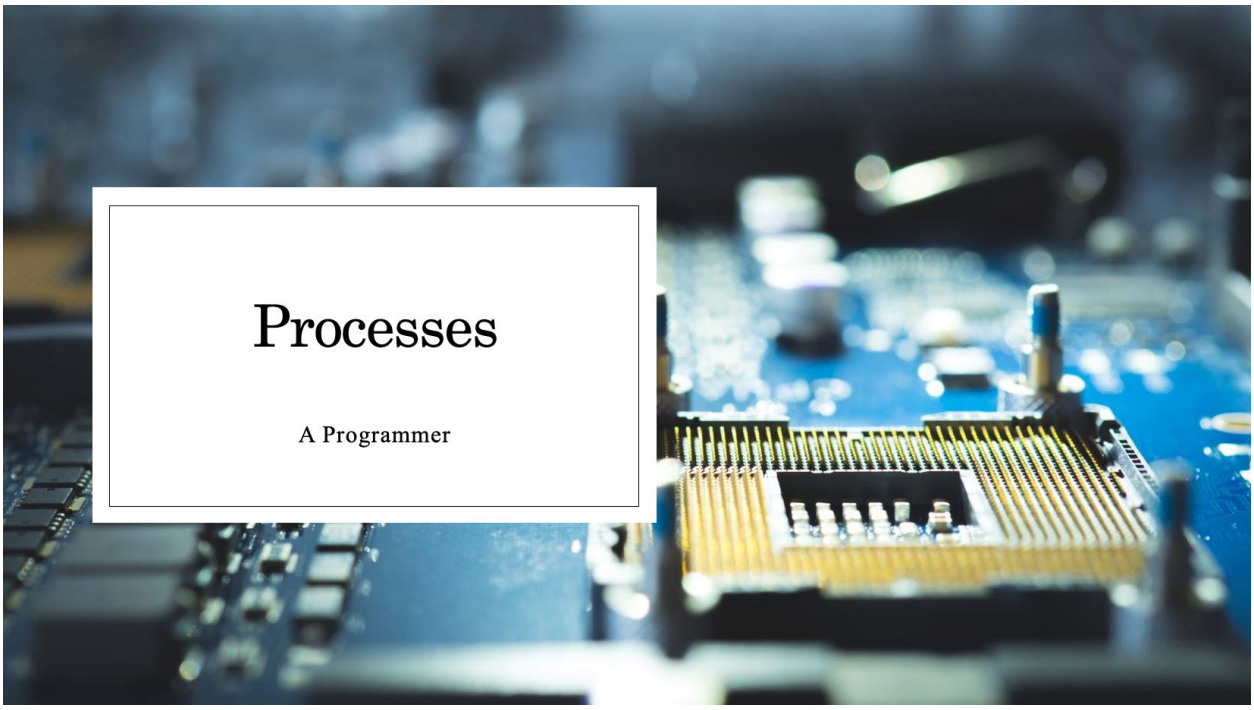
ပြောရရင် OS ဆိုတာကလဲ program တစ်ခုပဲ၊ အဲ့ဒီ OSဆိုတဲ့ program ထဲမှာမှ hardware resources တွေကို manage လုပ်ဖို့အတွက်ရော (ဥပမာ memory တွေ processor တွေကို manage လုပ်မယ့် program တွေဖြစ်တဲ့ traffic controller, memory management programs, I/O programs etc..) ၊ user တွေရဲ့ interaction တွေကို အဆင်ပြေပြေ operation လုပ်နိုင်ဖို့အတွက် (ဥပမာ input တွေကို လက်ခံပြီး output ပြန်ပြနိုင်အောင်လုပ်ပေးတာတို့) စသည်ဖြင့် အဲ့လို programs တွေပါ OS ဆိုတဲ့ program ထဲမှာတစ်ခါတည်းပါပြီးဖြစ်ပါတယ်။

ကျနော်တို့ OS ကို tree structure လိုခွဲချလိုက်မယ်ဆိုရင် Hardware resource တွေ အရင်ဆုံးရှိမယ် (CPU, Memory, I/O devices etc..)၊ hardware တွေပြီးရင် system programs တွေလာမယ် (compilers, loaders, OS etc..) ၊ system programs တွေရဲ့အပေါ်မှာမှ ကျနော်တို့နေ့စဉ်အသုံးပြုနေတဲ့ application programs တွေလာပါမယ်။

ဒီလောက်ဆိုရင်တော့ ယေဘုယျအားဖြင့် OS က ဘယ်လိုဆိုတာ သိလောက်ပြီလို့ထင်ပါတယ်။ ဘာလို့ OS ကို လေ့လာရလည်းဆိုတော့

ကျနော်တို့ရေးလိုက်တဲ့ code တွေ တစ်နည်းအားဖြင့် application program level ကနေလုပ်လိုက်တဲ့ input တွေမှာ system programs တွေထဲမှာ ဘယ်လို behavior နဲ့အလုပ် လုပ်သွားမယ်၊ system programs တွေကမှတစ်ဆင့် hardware resource တွေကို ဘယ်လို allocate လုပ်သွားမယ်ဆိုတာ သိသွားနိုင်မှာပါ။

Processes



Process ဆိုတာနဲ့ မိတ်ဆက်ပေးစရာတော့ တကူးတက မလိုလောက်ဘူးထင်ပါတယ်။ ရှင်းရှင်းနဲ့ ပြောရရင် program တစ်ခုကို run လိုက်တယ်ဆို process ဆိုတာဖြစ်လာတာပါပဲ။ code တွေရေးလိုက်တယ်၊ ပြီးရင် execute လုပ်လိုက်တယ်၊ အဲ့ကနေ process ဆိုတာဖြစ်လာပြီးတော့ ရေးထားတဲ့ program တွေထဲက code တွေအတွက် အလုပ်လိုက်လုပ်ပေးသွားတယ်။

တစ်ကယ်တန်းတော့ ဒီထက်ပိုနက်နဲပါတယ်။ system memory ထဲကနေ program တစ်ခု run လိုက်လို့ process ဆိုတာဖြစ်လာတာနဲ့အတူ အဲ့ဒီ process က memory ထဲမှာ Stack, Heap, Text, Data ဆိုပြီး အပိုင်း ၄ ပိုင်းခွဲပြီး အလုပ်လုပ်ပါတယ်။

Stack က temporary data တွေအတွက် (ဥပမာ function ထဲမှာ သိမ်းထားတဲ့ variable တို့, parameters တို့)။ Heap ကတော့ process run နိုင်ဖို့အတွက် allocate လုပ်ထားတဲ့ dynamic memory. Heap အကြောင်းကို data structures series မှာ ရေးထားပါတယ်။ Text ဆိုတာကတော့ program counter နဲ့ cpu registers တွေရဲ့လက်ရှိလုပ်နေတဲ့ activity value တွေအတွက်ဖြစ်ပါတယ်။ Data section ကတော့ global variable တွေနဲ့ static variables တွေအတွက်ဖြစ်ပါတယ်။

၄ပိုင်းခွဲပြီးအလုပ်လုပ်တာက ဟုတ်ပါပြီ။
ဘယ်လိုအလုပ်လုပ်လဲဆိုတာကရှိသေးတယ်။ program တစ်ခု run လိုက်ပြီးဆိုတာနဲ့ အလုပ်လုပ်သွားတဲ့ အဆင့် ၅ ဆင့်ရှိပါတယ်။ OS ပေါ်လိုက်ပြီး အစဉ်လိုက်က ကွဲပြားနိုင်တယ် ဆိုပေမဲ့လို့ အကြမ်းဖျင်းအားဖြင့်တော့ ဒီအစဉ်လိုက်ပါတယ်။

Start - process တစ်ခုကစမှတ် (process ဆိုပြီးဖြစ်လာတဲ့အချိန်ပေါ့)။

Ready – ဒီ stage မှာတော့ process က သူ့ကိုလာ assign လုပ်မယ့် processor ကိုစောင့်နေပါတယ်။ ပြီးတာနဲ့ allocate လုပ်ပြီး run လို့ရပြီပဲဖြစ်ပါတယ်။ ready stage ကိုရောက်လာရတဲ့ အကြောင်းအရင်းက နှစ်မျိုးရှိနိုင်ပါတယ်။ start stage

ပြီးလို့ရောက်လာတာရယ်၊ ဒါမှမဟုတ် process က running ဖြစ်နေပြီးတော့ scheduler တစ်ခုခုကြောင့် CPU က တစ်ခြား process တွေသွား run နေတဲ့အချိန် interrupt ဖြစ်သွားပြီး ready stage ပြန်ရောက်သွားတာမျိုးလဲဖြစ်နိုင်ပါတယ်။

Running – OS ရဲ့ scheduler တွေက processor ကနေ process တွေကို assign လုပ်လိုက်တဲ့အချိန်မှာ ဒီ stage ကိုရောက်ပါတယ်။

Waiting – process တွေက execute ထပ်လုပ်ဖို့အတွက်စောင့်ဖို့လိုလာတဲ့အချိန်ဆိုဒီနေရာရောက်ပါတယ်။ (ဥပမာ file တစ်ခုခုကိုလှမ်းဖတ်နေတာပဲဖြစ်ဖြစ်၊ user input ကိုစောင့်နေတာပဲဖြစ်ဖြစ်ပေါ့)။

Terminated - process ကသူ့ဘာသာ ပြီးသွားတာပဲဖြစ်ဖြစ်၊ OS က terminate လုပ်ပစ်လိုက်တာပဲဖြစ်ဖြစ် ဒီအဆင့်ကိုရောက်လာပါတယ်။ ပြီးတာနဲ့ main memory က process ကို remove လုပ်သွားပါတယ်။

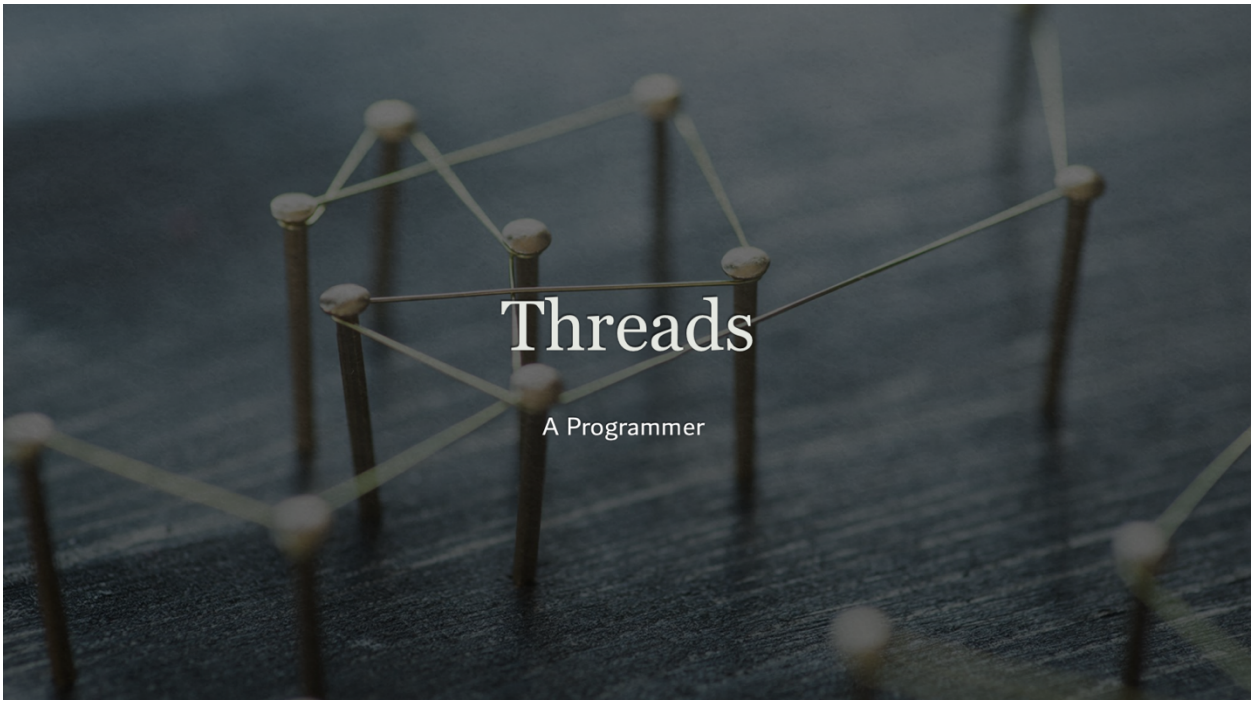
Process တွေကို run တိုင်းမှာ OS က PCB (Process Control Block) ဆိုတဲ့ data structure တစ်ခုနဲ့ process တွေကို track လုပ်ပါတယ်။ PCB ကိုတော့ integer value တွေနဲ့ identify လုပ်ထားပါတယ်။ PCB ထဲမှာဘာတွေပါလဲဆိုတော့

Process state - လက်ရှိ process က အပေါ်ကပြောထားတဲ့ ၅ ချက်ထဲကဘယ်အဆင့်ဆိုတာ။ Process privileges – system resources တွေကို allow လုပ်မှာလား။ disallow လုပ်မှာလား။ Process ID – unique ဖြစ်တဲ့ process id။ Pointer – parent process ကို link ထားတဲ့အရာ။ Program counter

- လာမယ့် instruction အတွက်ထောက်ထားတဲ့ pointer တစ်ခု။ CPU Scheduling info – process ရဲ့ priority level နဲ့ schedule လုပ်ရမယ့် အချက်အလက်တွေ။ CPU Registers – running stage မှာ process တွေကို store လုပ်ပေးမယ့် CPU registers တွေ။ Memory Management info – system memory ပေါ်မူတည်ပြီးတော့ OS ကအသုံးပြုသွားမယ့် memory limit တွေ page & segment table တွေ။ Accounting Information - လက်ရှိ process အတွက် အသုံးပြုမယ့် CPU amount နဲ့ execute လုပ်တဲ့ ID တွေ နဲ့ Input Output devices တွေရဲ့ status တွေစုထားတဲ့ IO status information တွေ ပါမှာဖြစ်ပါတယ်။

ဒီလောက်ဆိုရင်တော့ process တစ်ခုရယ်လို့ဖြစ်လာပြီဆို OS ကဘယ်လိုကိုင်တွယ်သွားလဲဆိုတာ ကောင်းကောင်းနားလည်သွားပြီပဲဖြစ်ပါတယ်။

Threads



အရင် article မှာတုန်းကတော့ process အကြောင်းကိုရေးသွားခဲ့ပါတယ်။ အဲ့ဒီ process ကနေပြီးတစ်ဆင့် execution ဖြစ်လာတဲ့ အရာတွေကို thread လို့ခေါ်ပါတယ်။ thread တိုင်းမှာ execute လုပ်ဖို့အတွက် program counter နဲ့ registers တွေပါဝင်ပါတယ်။ program counter , registers တွေအကြောင်းကို မသိသေးရင်တော့ processes ဆိုတဲ့ article ကိုအရင်သွားဖတ်သင့်ပါတယ်။

Process ထဲမှာရှိတဲ့ thread တွေက singly (တစ်ကြိမ်မှာ thread တစ်ခု) လည်း run လို့ရသလို multi-threaded (တစ်ကြိမ်တည်းမှာ thread တစ်ခုထက်မက) လည်း run လို့ရပါတယ်။ process တွေနဲ့ မတူတာက thread တွေက သူတို့အချင်းချင်းကို segment တွေ sharing လုပ်ထားလို့ရပါတယ်။ thread တစ်ခု က တစ်ခုခု change လုပ်လိုက်ပြီဆို တစ်ခြား thread တွေကို သိနိုင်ပါတယ်။ parallel architecture နဲ့ တစ်ပြိုင်နက်ထဲမှာတင် တစ်ခုထက်မက

run နိုင်တဲ့ အတွက် light weight ဖြစ်ပြီးတော့ OS ရဲ့ performance ကို improve ဖြစ်စေပါတယ်။

Process နဲ့ thread ကမတူပေမဲ့လည်း relation ရှိပါတယ်။
ကျနော်နားလည်ထားသလောက်ပြောရရင် process ဆိုတာက execution instance တစ်ခုအနေနဲ့ရှိတယ်။ thread တွေက အဲ့ဒီ execution instance ထဲက tasks တွေကို ဆောင်ရွက်ပေးမယ့် actual workers တွေဖြစ်ပါတယ်။ thread ကို real world example တစ်ခုပေးရရင် ကိုယ်က စာအုပ်တစ်အုပ်ဖတ်နေတယ်၊ ခနနားပြီး နောက်မှပြန်ဖတ်ချင်တယ်။ အခုဖတ်တဲ့နေရာကနေ ပြန်ဖတ်ချင်တယ်ဆို စာရွက် number, line number ကိုမှတ်ထားရပါမယ်။ (အဲ့လို process ကို thread ရဲ့ execution context လို့ခေါ်တယ်။ လက်ရှိမှာဆိုရင်တော့ execution context ထဲမှာ number နှစ်ခုပါပါတယ်၊ စာရွက် number ရယ် line number ရယ်)။ တစ်ချိန်ထဲမှာပဲ ကိုယ့်သူငယ်ချင်းတစ်ယောက်ကလဲ စာအုပ်ကိုယူဖတ်ပါတယ်။ သူဆီမှာလည်း ကျနော်လိုပဲ နားပြီးမှပြန်ဖတ်နိုင်ဖို့အတွက် execution context တွေရှိပါတယ်။ thread ရဲ့ အလုပ်လုပ်ပုံကလည်း ဒီပုံစံပဲ၊ computation တစ်ခုဆီတိုင်းမှာ execution context ရှိမယ်။ စာအုပ်ကို ကျနော့်သူငယ်ချင်းနဲ့ share လုပ်ပြီး ဖတ်နိုင်သလို tasks တွေအများကြီးကလည်း CPU ကို share လုပ်ပြီး consume လုပ်နိုင်ပါတယ်။

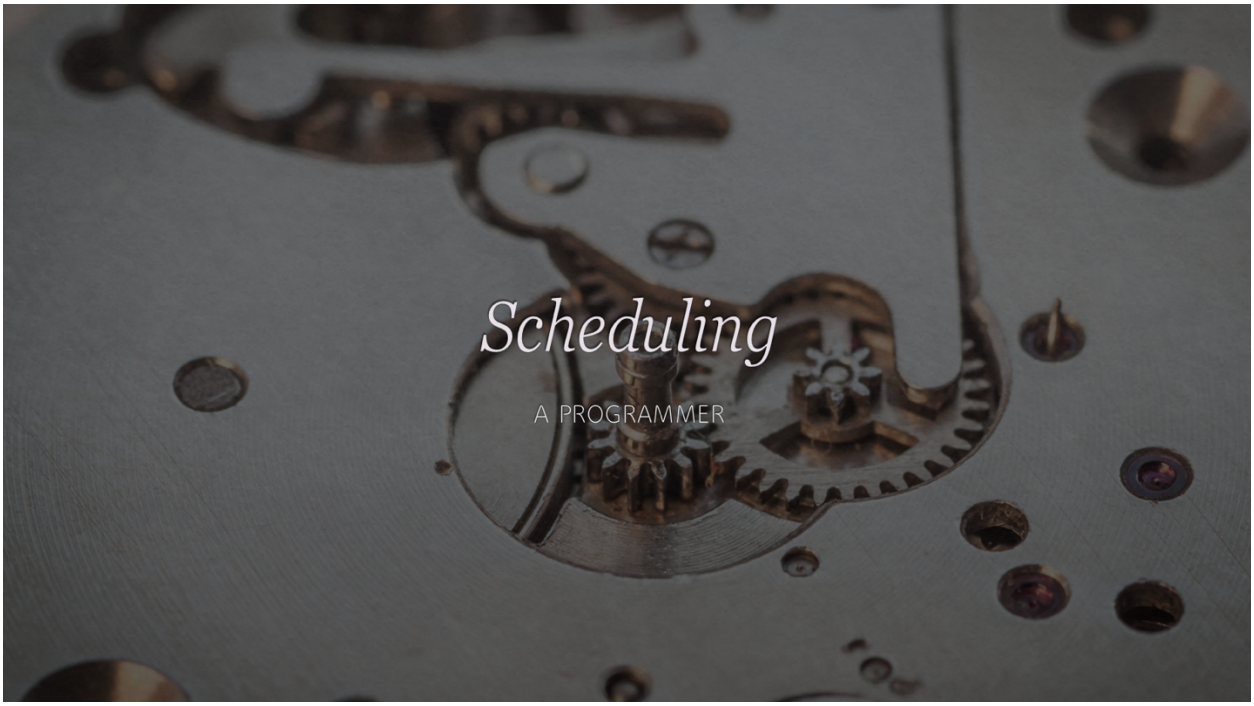
Process နဲ့ thread ရဲ့ အဓိက ခြားနားချက်က process က computation လုပ်ဖို့အတွက်လိုတဲ့ resource တွေပါတယ်။ thread ကတော့ execution context ပဲဖြစ်တယ်။

user level threads နဲ့ kernel level threads ဆိုပြီး Thread type ၂ မျိုးရှိတယ်။

user level မှာဖြစ်နေတဲ့ threads တွေကိုတော့ kernel က recognize မဖြစ်ဘူး၊ အဲ့ဒီ thread library မှာ thread တွေကို create/delete/schedule အစရှိတာတွေ လုပ်မယ့် code တွေပါတယ်၊ kernel level thread ထက်စာရင် user level thread တွေကပိုမြန်ပြီးတော့ OS မရွေးဘဲ run လို့ရပေမယ့်လို့ တစ်ချို့ OS တွေမှာ block ဖြစ်နိုင်တဲ့ case တွေရှိတယ်၊ ပြီးတော့ multi-processing မရပါဘူး။

Kernel level threads တွေကို OS က တိုက်ရိုက် support လုပ်ပေးထားပါတယ်။ threads တွေ management လုပ်ဖို့ကလဲ kernel ထဲမှာပဲရှိပါတယ်၊ user level application area မှာမရှိပါဘူး။ application တစ်ခုအတွက် process တစ်ခု ၊ အဲ့ process ထဲမှာပဲ threads တွေဖန်တီးပါတယ်။ process ထဲမှာရှိတဲ့ thread တစ်ခုခု block ဖြစ်သွားရင် အလားတူ thread နောက်တစ်ခုကို schedule ပြန်လုပ်ပေးနိုင်တယ်။ multi process architecture ကိုကောင်းကောင်းအလုပ်လုပ်နိုင်တယ်။ process တစ်ခုထဲမှာရှိတဲ့ thread တစ်ခုကနေ တစ်ခုကို control transfer လုပ်ရတဲ့ နေရာမှာတဲ့ Kernel ရဲ့ mode တစ်ခုကို လာပြီးတော့ switch လုပ်ပေးဖို့လိုအပ်ပါတယ်။

Scheduling



ပုံမှန် process manager တွေကလုပ်နေတဲ့ activity တွေဖြစ်တဲ့ process ကို remove လုပ်တာတို့ process တစ်ခုပြီးသွားရင် နောက်တစ်ခုကို select လုပ်ပြီး run တာတို့ကို process scheduling လို့ခေါ်ပါတယ်။ process scheduling က OS တစ်ခုမှာ multi processing ကိုလုပ်ပေးနိုင်ဖို့အတွက်အရေးကြီးတဲ့ အစိတ်အပိုင်းတစ်ခုဖြစ်ပါတယ်။ ဘာလို့လဲဆိုတာကို ဆက်ဖတ်ကြည့်လိုက်ရအောင်။

အရင် process article မှာတုန်းက PCB Block အကြောင်းတွေပြောထားပါတယ်၊ အဲ့ဒီ PCB တွေအားလုံးက OS က Process scheduling queue ထဲမှာသိမ်းထားပါတယ်။ ဘယ်လိုသိမ်းထားလည်းဆိုတော့ process တွေရဲ့ state ပေါ်မူတည်ပြီးသိမ်းပါတယ်၊ ဥပမာ waiting ဖြစ်နေတဲ့ process တွေဆို waiting queue, ready ဖြစ်နေတဲ့ process တွေဆို ready queue စသည်ဖြင့်အဲ့လို ခွဲပြီး

သိမ်းထားပါတယ်။ process တစ်ခုရဲ့ state ပြောင်းသွားပြီဆိုတာနဲ့ အဲ့ဒီ process ရဲ့ PCB ဟာလက်ရှိ ရှိနေတဲ့ queue ကနေ သူရဲ့ state နဲ့ သက်ဆိုင်တဲ့ queue ထဲကို move လုပ်သွားပါတယ်၊ ဥပမာ ထပ်ပေးရရင် ready ဖြစ်နေတဲ့ process ရဲ့ PCB က ready ဆိုတဲ့ queue ထဲမှာရှိမယ်၊ process ရဲ့ state က ready ကနေ running ဖြစ်သွားပြီဆို သူ့ရဲ့ PCB က ready queue ကနေ unlink လုပ်ပြီးတော့ running queue ဆီသွားသိမ်းပါတယ်၊ ဒီလိုသဘောတရားမျိုးဖြစ်ပါတယ်။

Queue တွေကို manage လုပ်ဖို့အတွက်ကို OS တွေကသုံးတဲ့ data structure တွေကလည်း ကွဲပြားပါတယ်။ ဥပမာ (FIFO, Round Robin, Priority Queue အစရှိသည်ဖြင့်) အသုံးပြုပြီးတော့ queue တွေကို manage လုပ်ပါတယ်။ Scheduling လုပ်တဲ့နေရာမှာ အဓိက process model ၂ ခုရှိပါတယ်။ Running & Not Running ဖြစ်ပါတယ်။ Running model ကတော့ ရှင်းတယ်၊ process တစ်ခုစလိုက်ပြီရှိတာနဲ့ running model အဖြစ်သတ်မှတ်လို့ရပါတယ်။ Not Running model မှာတော့ running လုပ်ဖို့စောင့်နေတဲ့ queue တွေရှိမယ်၊ အဲ့ဒီ queue တွေကိုတော့ linked list ကိုသုံးပြီး create လုပ်ထားတယ် linked list အကြောင်း ကျနော်ရေးထားတာကို ဒီမှာ သွားဖတ်ကြည့်လို့ရပါတယ်။ http://bit.ly/33LGBsA process တစ်ခုကို run နေရာကနေ interrupt လုပ်ခံရတာပဲဖြစ်ဖြစ်၊ အခုမှစပြီး run မလို့ပဲဖြစ်ဖြစ် waiting queue ထဲမှာရှိနေပါတယ်။ ပြီးသွားတဲ့ process တွေကိုတော့ discard လုပ်တယ်၊ ပြီးတာနဲ့ waiting queue ထဲက process တွေကို execute လုပ်ပါတယ်။

scheduler တွေရဲ့အလုပ်ကို summarise ပြန်လုပ်ရရင် သူတို့ရဲ့အဓိက အလုပ်က Job တွေကို select လုပ်ပြီး system ထဲမှာ execute လုပ်တာတို့ ဘယ် process ကို run မယ်ဆိုပြီး ဆုံးဖြတ်တာတို့ကိုလုပ်ပါတယ်။ ဒါပေမဲ့လည်း scheduler တွေကို system needs အရ သုံးမျိုးထပ်ခွဲထားပါသေးတယ်။ စာအရမ်းရှည်သွားမှာဆိုးတဲ့အတွက် ကျနော် ယေဘုယျပဲပြောသွားပါမယ်။

1. Long Term Scheduler ဒီကောင်ကိုတော့ Job scheduler လို့ခေါ်တယ်၊ ဘယ် programs တွေက process ဖြစ်လာဖို့စပြီး request လုပ်နေပြီလဲဆိုတာကို dertermine လုပ်ပေးပြီးတော့ queue ထဲက process တွေကို execute လုပ်ဖို့အတွက် memory ပေါ်တင်ပေးပါတယ်။
2. Short Term Scheduler CPU scheduler လို့ခေါ်သလို dispatcher လို့လဲခေါ်ပါတယ်။ ဘာလို့လဲဆိုတော့ ခုနက ပြောတဲ့ execute လုပ်ဖို့ memory ပေါ်ရောက်လာတဲ့ ကောင်တွေကို ready state ကနေ running state ကိုပြောင်းပြီး execute လုပ်ဖို့ CPU ထဲမှာ allocate လုပ်ပေးပါတယ်။
3. Medium Term Scheduler (Swapping) memory ပေါ်ကနေ process တွေကို remove လုပ်ဖို့အတွက်သုံးပါတယ်။ ဒါပေမဲ့ suspended ဖြစ်နေတဲ့ process (eg. IO requests တွေ)တွေလာရင်တော့ တန်းပြီး remove လုပ်လို့မရသေးပါဘူး။ memory ထဲမှာ နောက် process တွေအတွက်နေရာပေးပြီး suspend ဖြစ်နေတဲ့အတွက် ကောင်အတွက် secondary storage တစ်ခုထဲကို

move လုပ်ပေးထားပါတယ်။ အဲ့ဒါကို swapping လုပ်တယ်လို့ခေါ်ပါတယ်။

context switch ဆိုတဲ့ အရာအကြောင်းလေးပြောပြီး နိဂုံးချုပ်ပါမယ်။ scheduling အကြောင်းပြောပြီဆိုရင်တော့ context switch အကြောင်းတော့ မပါလို့မရပါဘူး။ multitasking လုပ်နိုင်ဖို့အတွက် context switch ကအရေးပါတဲ့အရာတစ်ခုဖြစ်ပါတယ်။ ဘာလုပ်ပေးလဲဆိုတော့ process တစ်ခုက state change ရတော့မယ်ဆို အဲ့ process ရဲ့context ကိုမှတ်ပေးထားပြီးတော့ လိုအပ်တဲ့အချိန်အဲ့ဒီ process ကိုပြန် run ပြီဆို မှတ်ထားတဲ့ context ကနေ resume ပြန်လုပ်ပေးနိုင်ပါတယ်။ လုပ်တဲ့ steps လေးတွေက ဒီလိုရှိပါတယ်။

- လက်ရှိ CPU မှာ run နေတဲ့ Process ရဲ့context ကို save လုပ်ပေးတယ်၊ အဲ့လို save လုပ်တာနဲ့အတူ process မှာရှိတဲ့ PCB block ထဲက တစ်ချို့ value တွေကိုလည်း update လုပ်ပေးပါတယ်။ (P1 လို့ မှတ်ထားလိုက်ပါမယ်)
- ပြီးတာနဲ့ သူပြောင်းမယ့် state ကိုပြောင်းပြီးတော့ relevant ဖြစ်တဲ့ queue ထဲကိုရောက်သွားတယ်။
- နောက် process တစ်ခုကို execute ထပ်လုပ်တယ်။ (P2 လို့မှတ်ထားလိုက်ပါမယ်)
- P1 ရဲ့ no 1 and 2 ကအဆင့်တွေလို့ပဲလုပ်ပါတယ်။ ဒီနေရာမှာ memory management data တွေကို update လုပ်ဖို့လိုအပ်လာရင်လည်းလုပ်ပါတယ်။

- P1 ကို ပြန် execute လုပ်တော့မယ်ဆို မှတ်ထားတဲ့ context တွေ၊ PCB block က data တွေကို reference ပြန်လုပ်ပြီး resume ပြန်လုပ်ပေးနိုင်ပါတယ်။

ဒီလောက်ဆိုရင်တော့ Scheduling ဘယ်လိုအလုပ်လုပ်တယ်ဆိုတာအပြင် သူ့ထဲမှာ ပါတဲ့ modules တွေရဲ့လုပ်ဆောင်ချက်ကို ကောင်းကောင်းသဘောပေါက်သွားပြီပဲဖြစ်ပါတယ်။

Memory Management Part 1

computer ထဲမှာရှိတဲ့ internal physical memory ကို main memory လို့ခေါ်ပါတယ်။ main memory ဆိုပြီးခေါ်ရတဲ့အကြောင်းရင်းက external memory storage ဖြစ်တဲ့ disk drives တွေနဲ့ ရောသွားမှာဆိုးလို့ အရှေ့မှာ main ထည့်ပြီးခေါ်ရခြင်းဖြစ်ပါတယ်။ main memory ကို RAM လို့လည်းခေါ်လို့ရပါတယ်။ computer က main memory ထဲမှာဖြစ်လာတဲ့ data တွေကိုပဲ manage လုပ်လို့ရပါတယ်၊ ဒါကြောင့်မို့လို့ programs တွေ runလိုက်ပြီဆို storage devices တွေကနေပြီးတော့ main memory ဆီကိုလာရပါတယ်။ ဒါမှသာ computer က အဲ့ဒီ process ကို manage လုပ်လို့ရမှာပါ။

memory management မှာအရေးပါတဲ့ module တစ်ခုခြင်းဆီကိုအောက်မှာထပ်ရေးပေးသွားပါမယ်။

Process Address process တစ်ခုက code တွေဆီကနေ reference ယူထားတဲ့ logical address (memory cell, storage element, network host etc.) တွေရှိပါတယ်။ program တစ်ခု memory allocate လုပ်တဲ့အချိန်မှာ OS က logical address တွေကို memory management unit (MMU) ကိုသုံးပြီး physical address အဖြစ် map လုပ်ပါတယ်။ အသေးစိတ်ရှင်းရမယ်ဆို logical address က program တစ်ခု run လိုက်တဲ့အချိန်မှာ CPU က generate လုပ်ပေးထားတာဖြစ်ပြီး physically မရှိဘဲ virtually ပဲရှိပါတယ်။ physical address ကတော့ memory ထဲမှာကို physical location နဲ့ရှိပါတယ်။ MMU ကတစ်ခုနဲ့တစ်ခုကို correspond လုပ်နိုင်ဖို့အတွက် ကြားခံ computation လုပ်ပေးတဲ့အရာတစ်ခုလို့ပြောလို့ရပါတယ်။ Okay , OS က memory allocate နေရာကိုပြန်သွားရအောင်။ allocate လုပ်တဲ့နေရာမှာ address type ၃ခုကိုသုံးပြီးလုပ်ပါတယ်။

symbolic address

- source code ထဲမှာသုံးတဲ့ address ဖြစ်ပါတယ်။ variable name တို့ constant တို့ အစရှိသဖြင့်ပါပါတယ်။

Relative address

- program ကို compile လုပ်တဲ့အချိန်မှ compiler က symbolic address တွေကို relative address အဖြစ်ပြောင်းလဲပေးလိုက်ပါတယ်။

Physical address

- နောက်ဆုံး program က memory ထဲမှာ load လာလုပ်တဲ့အချိန်မှ loader က physical address တွေကို generate လုပ်ပေးလိုက်ပါတယ်။

Static , Dynamic loading and Static , Dynamic Linking program တွေက execute လုပ်ဖို့အတွက် main memory ထဲကိုလာလုပ်ကြရပါတယ်။ တစ်ချို့ program တွေက main memory ထဲမှာပဲ completely တစ်ခါတည်း run သွားတယ်။ အဲ့ဒါကို static loading လို့ခေါ်ပါတယ်။ တစ်ချို့ကျတော့ main memory ပေါ်မှာ partially ပဲလာrun ပြီးကျန်တဲ့ libraries တွေ modules တွေကို disk ပေါ်မှာထားထားပါတယ်။ အဲ့ဒီ modules တွေကိုလိုအပ်လာတဲ့အချိန်မှာသာ main memory ထဲမှာ load လုပ်ပါတယ်။ ဒါမျိုးကိုတော့ Dynamic loading လို့ခေါ် ပြီးတော့ system အတွက် efficiency ပိုကောင်းစေတယ်လို့ပြောလို့ရပါတယ်။ main memory ပေါ်မှာ တစ်ခါတည်းနဲ့ completely compile လုပ်သွားတဲ့ program ကို static linking နဲ့လုပ်သွားတယ်လို့ပြောလို့ရပါတယ်။ ဘာလို့လဲဆိုတော့ တစ်ခြား ဘာ modules တွေကိုမှ လှမ်းဆွဲယူစရာမလိုဘဲ single program တည်းနဲ့ပဲ compile လုပ်လို့ရအောင် linking လုပ်ထားတဲ့အတွက်ပါ။ Dynamic loading နဲ့သွားတဲ့ program ကတော့ modules တွေအကုန်လုံး တစ်ခါတည်း link မလုပ်ထားဘဲနဲ့ လိုတဲ့အချိန်မှသာ link လုပ်တဲ့အတွက် dynamic linking လုပ်တယ်လို့ခေါ်ပါတယ်။

Memory Management Part2

Swapping swapping အကြောင်းကိုတော့ scheduling article မှာလည်းပြောသင့်သလောက်တော့ ပြောထားပြီးပါပြီ။ main memory ထဲမှာလက်ရှိ run နေတဲ့ process ကို secondary storage တစ်ခုထဲကို move လုပ်ထားပြီး အချိန်တစ်ခုကြာမှ main memory ထဲကို ပြန်ထည့်ပြီး run တယ်။ (ဘာလို့ move လုပ်ရလဲဆိုတာကတော့ run ရမယ့် process ရဲ့ memory ကကြီးလို့မဆန့်လို့ခွဲထုတ်တာလိုဖြစ်နိုင်သလို process က စောင့်ရမယ့် task တွေပါလာမယ်ဆိုရင်လဲ secondary storage ထဲကို move လုပ်ပြီးစောင့်ပါတယ်) ။ swapping လုပ်တာက performance ကိုထိခိုက်စေနိုင်ပေမဲ့ တစ်ဖက်ကကြည့်ရင်လည်း multiprocessing ကိုလုပ်နိုင်နေပါတယ်။ တစ်နည်းအားဖြင့် swapping လုပ်တာကို memory compaction လို့လဲခေါ်ပါတယ်။

Memory allocation main memory မှာ OS ရှိတဲ့ Low memory နဲ့ user processes တွေကိုလုပ်တဲ့ high memory ဆိုပြီး partition နှစ်မျိုးရှိပါတယ်။ Memory allocate လုပ်တဲ့နေရာမှာလည်း single partition နဲ့ multiple partition allocation ဆိုပြီးခွဲထားပါတယ်။

single partition allocation user processes တွေကြောင့်ဖြစ်လာတဲ့ side effect တွေကြောင့် OS ရဲ့ source code နဲ့ data တွေကို changes လာဖြစ်စေနိုင်ပါတယ်၊ဒါပေမဲ့ OS ကို အဲ့ဒီ threats တွေအတွက် protect လုပ်ထားပြီးသားဖြစ်ပါတယ်။ ထိုနည်းအတူ user processes အချင်းချင်းကိုလည်း changes တွေမဖြစ်အောင် protect လုပ်ထားဖို့လိုပါတယ်။ အဲ့အတွက်ကြောင့်မို့လို့ relocation registers တွေကိုကိုသုံးပါတယ်။ relocation register မှာတော့ အသေးဆုံးဖြစ်တဲ့ physical address ပါမယ်၊ limit register ထဲမှာတော့ logical address range တစ်ခုပါပါမယ်။ relocation နဲ့ limit registers တွေ အရ logical address တိုင်းဟာ limit register ထက်ငယ်နေရပါမယ်။ MMU(memory management unit) က relocation register ထဲက logical address တွေကို map လုပ်ပါတယ်။ map လုပ်ထားတဲ့ address တွေကိုမှ memory ထဲကိုနောက်ဆုံးအနေနဲ့ပို့လိုက်ပါတယ်။

multiple partition allocation (contiguous memory allocation) memory ကို fixed size partitions လေးတွေအဖြစ်ပိုင်းထားပါတယ်။ contiguous allocation မှာ process တစ်ခုကို အဲ့ဒီ ပိုင်းထားတဲ့ memory block တစ်ခုအပေါ်မှာ allocate လုပ်ပါတယ်။ partition တစ်ခုကို process တစ်ခုပုံစံမျိုးပေါ့။ partition တစ်ခုက free ဖြစ်နေတဲ့အချိန်မှာ process တစ်ခုကို queue ထဲကနေ select လုပ်ပြီး

allocate လုပ်ပါတယ်။ အဲ့လို free ဖြစ်နေတဲ့ partition block လေးတွေကို holes လို့လဲခေါ်ပါတယ်။ process ပြီးသွားတဲ့အချိန်မှာ partition free up ဖြစ်ပြီးတော့ နောက် process တွေကိုဆက်ပြီး allocate လုပ်သွားပါတယ်။

Fragmentation Memory ထဲမှာ processes တွေက load လာလုပ်လိုက် remove လုပ်သွားလိုက်လုပ်နေကြပါတယ်။ အဲ့လိုလုပ်ပါများလာတဲ့အခါ free ဖြစ်နေတဲ့ memory space တွေက အပိုင်းသေးသေးလေးတွေအဖြစ် ပြန့်ကျဲသွားပါတယ်။ process တစ်ခု allocate လာလုပ်တော့မယ့်အချိန်မှာ memory block လေးတွေကသေးနေတဲ့အတွက် အသုံးမဝင်တော့တာမျိုးဖြစ်တာကို fragmentation ဖြစ်သွားတယ်လို့ခေါ်ပါတယ်။ fragmentation မှာ နှစ်မျိုးရှိပါတယ်။ Internal fragmentation process တစ်ခုကို allocate လုပ်ပေးမယ့် memory က လုပ်ပေးရမယ့် memory ထက်ပိုပြီးတော့ ရှိနေပါတယ်။ allocate လုပ်ပြီးတဲ့အခါမှာ memory အပိုတွေထွက်လာပေမဲ့လည်း တစ်ခြား process တွေအတွက် allocate မလုပ်ပေးနိုင်ပါဘူး။

External fragmentation Process တစ်ခုကို allocate လုပ်ပေးဖို့လုံလောက်ပေမဲ့လည်း contiguous မဖြစ်တဲ့အတွက်ကြောင့်လုပ်ပေးလို့မရပါဘူး။

အဲ့ဒီ fragmentation ဖြစ်တာတွေကိုကာကွယ်နိုင်ဖို့အတွက် Paging ဆိုပြီး technique တစ်ခုရှိပါတယ်။ virtual memory တွေကို ကောင်းကောင်းကစားနိုင်ဖို့အတွက် paging technique ကအရေးပါပါတယ်။ paging က fragmentation ကိုဘယ်လိုကာကွယ်လဲဆိုတော့ process address

space ကိုတူညီတဲ့ partition blocks တွေအဖြစ်ခွဲချလိုက်ပါတယ်။ ဥပမာ process တစ်ခုလာပြီဆို တစ်ခါတည်း ပစ်သွင်းလိုက်မယ့်အစား process က 1000 bytes ယူမယ်ဆို 100bytes အဖြစ် block 10 ခုခွဲချလိုက်ပါတယ်၊ ဒါက ဥပမာပြတာပါ။ size က (2 power ဖြစ်ပြီး 512bytes to 8192 bytesအတွင်းရှိနိုင်ပါတယ်)။ အဲ့လိုblocks လေးတွေကို pages တွေလို့ခေါ်ပါတယ်။ process address space အတိုင်းပဲ main memory ကိုလဲ pages တွေအတိုင်းခွဲချလိုက်ပြီးတော့ အလုပ်လုပ်လိုက်ခြင်းအားဖြင့် fragmentation ကို avoid လုပ်ပါတယ်။

Segmentation Segmentation ဆိုတာကလဲ paging လိုပဲ memory management လုပ်တဲ့ technique နောက်တစ်ခုဖြစ်ပါတယ်။ paging နဲ့မတူတဲ့အချက်က paging က fixed size တွေအဖြစ်ခွဲပေမယ့် segmentation က size တွေက variable ဖြစ်ပါတယ်။ segment တိုင်းမှာလုပ်ဆောင်ရမယ့် program ရဲ့ functions တွေ data structures တွေနဲ့ တစ်ခြား utility functions တွေလဲပါပါတယ်။ segment တိုင်းအတွက်ကို os က segment map table တစ်ခုထားပြီးတော့ အလုပ်လုပ်ပါတယ်။ table ထဲမှာ segment number, segment size, memory location address တွေပါပါတယ်။

အခုပြောသွားတဲ့ paging မှာရော segmentation မှာရောကောင်းတဲ့အချက်တွေရော ဆိုးတဲ့အချက်တွေပါရှိပါသေးတယ်။ အဲ့ဒါကိုတော့ တစ်ချက် research ထပ်လုပ်သင့်ပါတယ်။

အခုလောက်ဆိုရင်တော့ memory management အပိုင်းက module တော်တော်များများကို သိသွားပြီပဲဖြစ်ပါတယ်။ ဒါပေမဲ့

ဒီထက်ပိုပြီးနက်နဲတဲ့အပိုင်းတွေရှိသေးတဲ့အတွက်
မိမိဘာသာဆက်လက်ပြီးလေ့လာကြပါဦးလို့ တိုက်တွန်းလိုက်ပါတယ်။

Inter Process Communication

Process တစ်ခုကိုခွဲချလိုက်ပြီဆို Independent ဖြစ်တဲ့ process နဲ့ co-operating ဖြစ်တဲ့ Process ဆိုပြီး နှစ်မျိုးရှိပါတယ်။ မှတ်ရတာတော့ လွယ်တယ်၊ independent process က အခြားသော process တွေ execution ဖြစ်နေတဲ့အချိန်မှာသူ့ကိုဘာမှလာ affect မဖြစ်ဘူး၊ co-operating ကတော့ဖြစ်တယ်။ သူ့တို့ရဲ့သဘောတရားကိုကြည့်ပြီးတော့၊ သြော် independent process က system အတွက်ပိုကောင်းနိုင်တယ်လို့ထင်ရင်မှားမယ်။ တစ်ကယ်တမ်းတော့ co-operating process က ပိုပြီးတော့ system ရဲ့ computation speed အတွက်ရော တစ်ခြားသော efficiencies

တွေကိုပါပိုကောင်းစေပါတယ်။ ဘာအကြောင်းတွေကြောင့်လဲဆိုတာ ဆက်ဖတ်ကြည့်ရအောင်။

IPC လို့ခေါ်တဲ့ inter process communication က process တစ်ခုနဲ့တစ်ခု ချိတ်ဆက်လို့ရအောင် ပြီးတော့ သူတို့ရဲ့ actions တွေကို sync လုပ်လို့ရအောင် ပြုလုပ်ပေးနိုင်တယ်။ အဲ့လို ချိတ်ဆက်တာကို co-operation ရဲ့ method လို့ ယူဆလို့ရတယ်။ Process တွေတစ်ခုနဲ့တစ်ခုချိတ်ဆက်တဲ့နေရာမှာ shared memory နဲ့ message parsing ဆိုပြီး နည်းနှစ်မျိုးနဲ့ ချိတ်ဆက်နိုင်ပါတယ်။

Shared Memory Method Process 1 နဲ့ process 2 ရှိတယ်ဆိုကြပါစို့။
နှစ်ခုလုံးက တစ်ပြိုင်နက်တည်းမှာ execute လုပ်နေတယ်၊ process တစ်ခုနဲ့ တစ်ခုလည်း resource တွေလည်း share လုပ်ရင်းနဲ့ပေါ့။
ဘယ်လိုလုပ်လဲဆိုတော့ Process 1 က သူ execute လုပ်ခဲ့ရတဲ့ အချိန်က resources တွေ၊ computations တွေနဲ့ ပတ်သတ်တဲ့ အချက်အလက်တွေကို generate လုပ်ပြီးတော့ shared memory ထဲမှာ record တွေမှတ်ခဲ့တယ်။ process 2 က အဲ့ဒီ shared လုပ်ထားတဲ့ Information တွေကိုသုံးချင်ပြီဆို shared memory ထဲမှာ record လုပ်ထားတာရှိမရှိကြည့်တယ်၊ ရှိတယ်ဆို လှမ်းယူသုံးလိုက်တယ်။ ဆိုတော့ shared memory method ကတော့ရှင်းတယ်။ process တွေ shared memory ကိုသုံးပြီး information တွေလှမ်းယူလို့ရတယ်။ ကိုယ့်မှာရှိတဲ့ Information တွေကိုလည်း တစ်ခြား process တွေအတွက် shared memory ထဲမှာ record လုပ်ပေးထားလို့ရတယ်။ ပြောရမယ်ဆိုရင် producer နဲ့ consumer လိုပဲပေါ့။ producer ကထုတ်ပေးတယ်၊ consumer

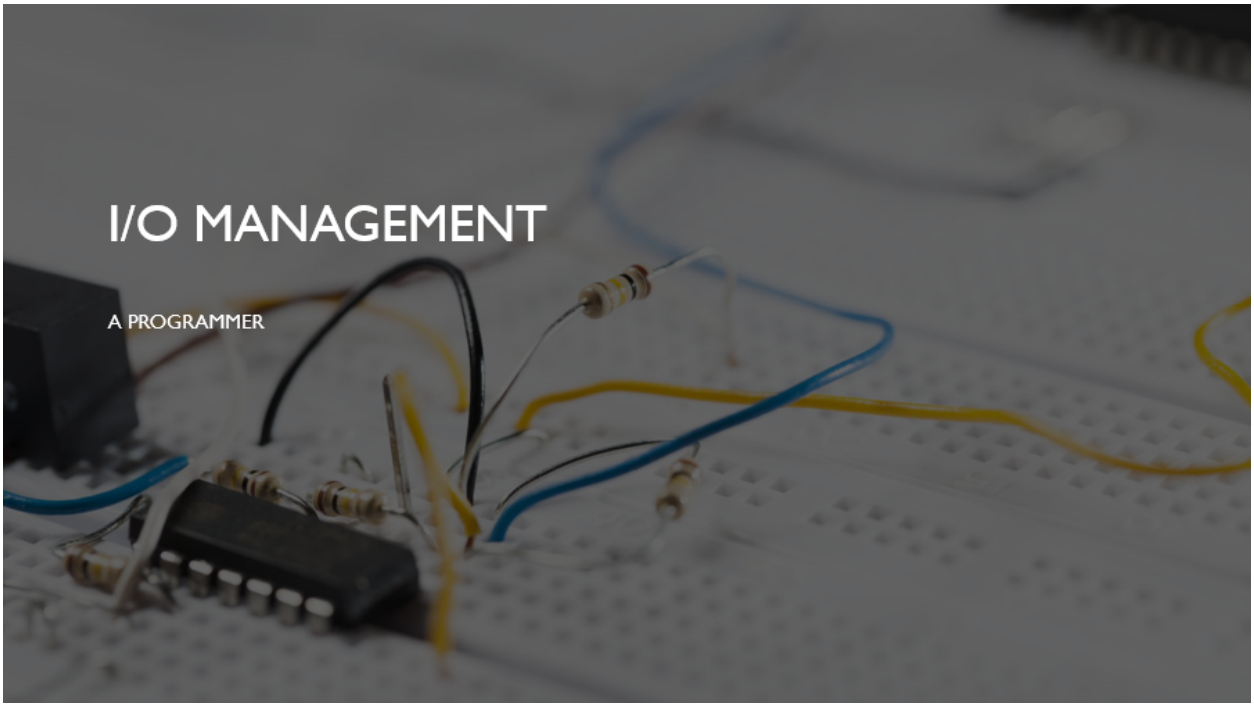
ကသုံးပေးတယ်။ သူတို့နှစ်ခုကြားမှာ shared memory ကိုက bridge တစ်ခုအနေနဲ့ လုပ်ဆောင်ပေးတယ်။

Message Parsing Method အခု method မှာကတော့ shared memory တွေဘာတွေသုံးစရာမလိုတော့ဘဲနဲ့ communicate လုပ်ကြပါတယ်။ ဘယ်လိုလုပ်ကြလဲဆိုတော့ communication link တစ်ခုထုတ်ပေးပါတယ်၊ အဲ့အပေါ်ကတော့ communicate လုပ်ကြပါတယ်။ အဲ့ link ကရှိပြီးသားဆိုရင်တော့ ထပ်ထုတ်စရာမလိုတော့ဘူးပေါ့။ communicate လုပ်တဲ့ ပုံစံကလည်းရှင်းတယ်။ basic primitives ပုံစံလေးတွေပဲသုံးပြီးတော့ လုပ်သွားတာ။ primitive နှစ်ခုရှိမယ်၊ send လုပ်မယ်၊ receive လုပ်မယ်။ Send လုပ်တဲ့အချိန်မှာ param နှစ်ခုထည့်ပြီး send လုပ်လို့ရတယ်။ (message ရယ်၊ ပို့မယ့် destination ရယ်(optional)) Receive လုပ်တဲ့အချိန်မှာလည်း param နှစ်ခုသုံးလို့ရတယ်၊ (message ရယ်၊ လက်ခံရယူမယ့် host name ရယ်(optional)) Message size ကတော့ fixed လည်းဖြစ်နိုင်သလို vary လည်းဖြစ်နိုင်ပါတယ်။ fixed or variable ပေါ်မူတည်ပြီး OS designer နဲ့ programmer ရဲ့လုပ်ရမယ့်အပိုင်းတွေကလည်း ကွဲပြားသွားမှာပါ၊ အဲ့အပိုင်းကိုတော့ ကျနော် အကျယ်မချဲ့တော့ပါဘူး။ အိုကေ ဒါဆိုရင် message ကို ထပ်ပြီး analyze လုပ်ကြည့်မယ်ဆို သူ့မှာ header အပိုင်းနဲ့ body အပိုင်းဆိုပြီး ထပ်ကွဲသေးတယ်။ header အပိုင်းမှာတော့ message type,length တို့ source and destination id တို့နဲ့ တစ်ခြား control information တွေလည်းပါပါတယ်။ control information ဆိုတာကတော့ sequence တို့ priority တို့ကိုသိမ်းထားတာပေါ့။ body အပိုင်းမှာတော့ ပုံမှန်အတိုင်း message ရဲ့ content ပါတယ်။

ပုံမှန်အားဖြင့်ဆိုရင်တော့ message တွေက FIFO structure နဲ့
အလုပ်လုပ်ပါတယ်။

ဒါဆိုရင်တော့ IPC အကြောင်းတောတော်လေးသိသွားမယ်လို့ထင်ပါတယ်။
တစ်ကယ်တော့ မဟုတ်ပါဘူး :3၊ စတာပါ ယေဘုယျအားဖြင့် လောက်တော့
တော်တော်လေးသိသွားပြီလို့ထင်ပါတယ်။ တစ်ကယ်တမ်း ထပ်ပြီး detail dive
လုပ်လို့ရပါသေးတယ်။ ဥပမာ shared memory တို့ message parsing တို့ ရဲ့
အလုပ်လုပ်ပုံ အသေးစိတ်တွေကို ထပ်ပြီးတော့
လေ့လာကြည့်လို့ရပါသေးတယ်။

I/O Management



I/O devices တွေကို manage လုပ်ဖို့ဆိုတာကလည်း OS ရဲ့အရေးကြီးတဲ့ အပိုင်းတစ်ခုဖြစ်ပါတယ်။ I/O device ဆိုတာကတော့ အားလုံးသိတဲ့အတိုင်း keyboard , mouse , drivers, usb devices စတာတွေပေါ့။ စဉ်းစားကြည့်မယ်ဆို I/O system ရဲ့အလုပ်လုပ်ပုံက အရမ်းမရှုပ်ပါဘူး။

- I/O request တွေကို system ကနေယူမယ်။
- ရလာတာတွေကို I/O device တွေဆီကိုပို့ပေးမယ်။
- Device တွေကနေပြန်ရလာတဲ့ response ကိုယူပြီး application level ဆီကိုပို့ဖို့ အလုပ်လုပ်မယ်။ I/O devices တွေထဲမှာ data တွေကို block လိုက်ပို့တဲ့ block device နဲ့ character တစ်ခုခြင်းဆီလိုက်ပို့တဲ့ character device ဆိုပြီးကွဲပါသေးတယ်။ block device ဆိုရင် data တွေ communicate လုပ်တဲ့အချိန် block လိုက်ပို့တဲ့ကောင်တွေ၊ ဥပမာ

hard drive တို့ ဘာတို့ ၊ character လိုက်ပို့တဲ့ကောင်တွေကတော့ system ထဲမှာရှိတဲ့ console devices တွေ serial devices တွေပေါ့။

နောက်ပြီးတော့ ကျနော်တို့ driver တွေအကြောင်းကြားဖူးမယ်ထင်တယ်။ laptop ဝယ်ရင် driver ခွေတွေဘာတွေပေးတာတို့ ဒါမှမဟုတ် driver လိုနေလို့ driver သွင်းရတာတို့။ I/O device တိုင်းမှာလည်း driver တွေလိုတယ်။ ဥပမာ touch pad driver မရှိဘူးဆို touch pad ကသုံးလို့ရမှာမဟုတ်ဘူး။ ဒါကြောင့် OS က I/O devices တွေကို handle လုပ်ဖို့ဆိုရင် driver တွေရဲ့အကူအညီကိုသုံးပြီး handle လုပ်ပါတယ်။

ကျနော်အပေါ်မှာ data communicate လုပ်တဲ့ဆိုတဲ့အကြောင်းလေးပါခဲ့တယ်။ ဆိုတော့ system CPU က အဲ့ဒီ I/O devices တွေဆီကို ဘယ်လိုနည်းတွေနဲ့ communicate လုပ်လဲဆိုတာမေးစရာရှိပါတယ်။ communicate လုပ်တဲ့ instructions ပုံစံသုံးမျိုးရှိပါတယ်။

Special Instruction I/O သူ့ကိုဘာလို့ special instruction လို့ခေါ်လည်းဆိုတော့ I/O devices တွေကို control လုပ်ဖို့ကို CPU ဆီက Instructions တွေနဲ့တိုက်ရိုက်သုံးပြီးလုပ်လို့ပဲ။ အဲ့ဒီ Instructions တွေအတိုင်း I/O devices တွေဆီကနေ data တွေ communicate လုပ်ဖို့ကိုဆောင်ရွက်ပါတယ်။

Memory-mapped I/O ဒီအပိုင်းမှာတော့ data communication လုပ်ဖို့ကို CPU ကနေတိုက်ရိုက်မသွားတော့ဘူး၊ သတ်မှတ်ပေးထားတဲ့ memory space ပေါ်ကနေပဲ communicate လုပ်သွားတယ်။ အဲ့ဒီ memory space အတွက်ကိုလည်း OS က allocate လုပ်ပေးထားတာ၊ ဆိုလိုချင်တာက CPU နဲ့

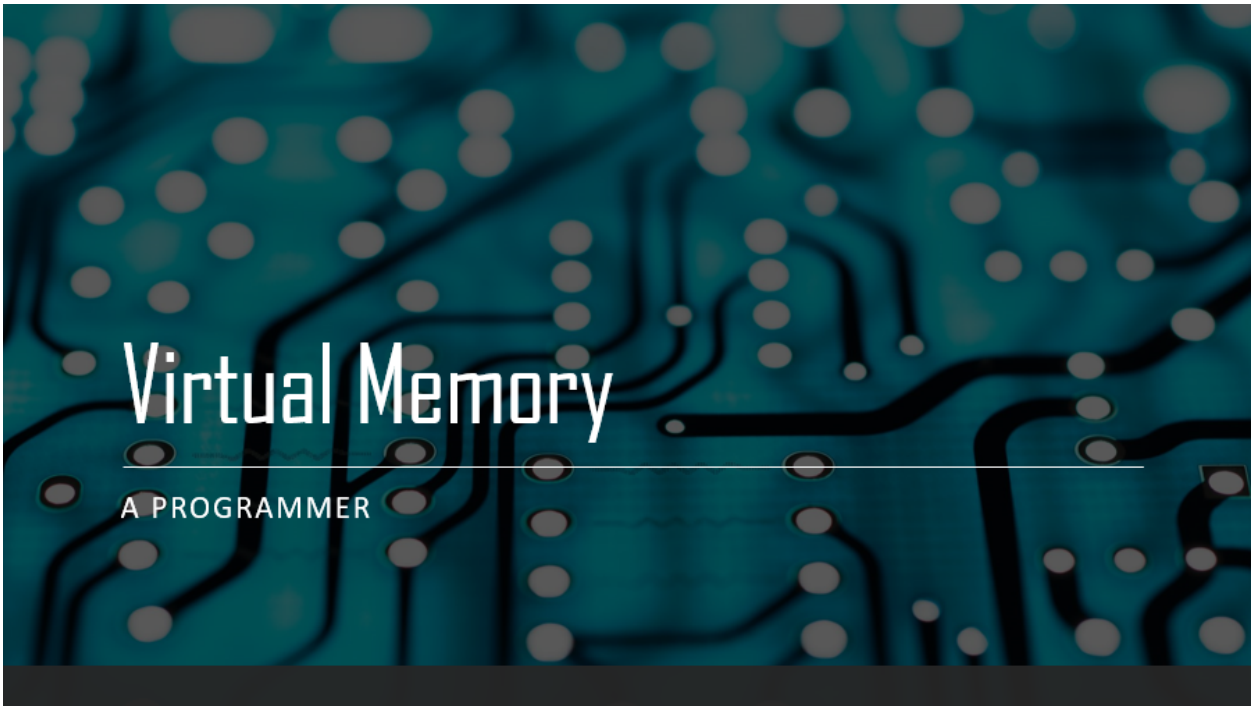
I/O devices တွေကြားထဲမှာ memory space တစ်ခုခံထားပြီး data တွေ transfer လုပ်တယ်။ အဲ့လို transfer လုပ်တဲ့အချိန်မှာလည်း I/O devices တွေက CPU နဲ့ကို async ပုံစံနဲ့ communicate လုပ်တယ်။ process ပြီးပြီဆိုရင် CPU လည်း stop ဖြစ်သွားတယ်။ memory-mapped I/O ကို high speed transfer လုပ်တဲ့ devices တွေမှာသုံးတယ်၊ ဥပမာ disk drives တွေလိုကောင်တွေ။

Direct Memory Access DMA လို့လည်းခေါ်တယ်။
အပေါ်မှာကျနော်ပြောထားတဲ့ memory-mapped က high speed devices တွေအတွက်ဆိုပေမယ့် စဉ်းစားစရာရှိတာက keyboard တို့လို slow speed devices တွေကျတော့ ဘယ်လိုလုပ်မလဲပေါ့၊ သူတို့က byte တစ်ခုခြင်းဆီလွှတ်နေတာ၊ ဆိုတော့ CPU ကအဲ့ဒါကိုထိုင်စောင့်ပြီး အလုပ်လုပ်နေရင် waste အများကြီးဖြစ်သွားတယ်။ အဲ့နေရာမှာ အခုပြောမယ့် DMA ဆိုတာဝင်လာတာပဲ။ CPU က I/O devices တွေနဲ့ communicate လုပ်ဖို့အတွက် DMA ကို ခွင့်ပေးလိုက်တယ်။ communication လုပ်တဲ့အချိန်မှာ devices တွေနဲ့ DMA တိုက်ရိုက်လုပ်ပစေ၊ သူဝင်မပါတော့ဘူးဆိုတဲ့ သဘော (include main memory between DMA and devices tho)။ လိုအပ်တဲ့ authorities မှန်သမျှကိုလဲ DMA ကို grant လုပ်ပေးထားတယ်။
သူဝင်ပါမယ့်အချိန်က communication စလုပ်မယ့်အချိန်ရယ် အကုန်လုပ်ပြီးသွားလို့ end ဖြစ်သွားတဲ့အချိန်ပဲဝင်ပါတော့မယ်။ DMA ကလည်း data transfer လုပ်တဲ့နေရာမှာ manage လုပ်နိုင်ဖို့အတွက် DMA controller ဆိုတာကိုသုံးသေးတယ်။ အဓိက ကတော့ data transfer လုပ်တဲ့နေရာမှာ handle လုပ်နိုင်ဖို့ပဲ၊ ဥပမာ data transfer လုပ်တဲ့နေရာမှာ source & destinations သတ်မှတ်တာတို့၊ transfer ဖြစ်သွားတဲ့ bytes တွေ count

ထားတာတို့ စသည်ဖြင့် transfer လုပ်တဲ့နေရာမှာ လိုအပ်တာတွေကို handle လုပ်ပေးတယ်။

ဒီလောက်ဆိုရင်တော့ I/O devices တွေ OS ပေါ်မှာဘယ်လိုအလုပ်လုပ်နေလဲဆိုတဲ့ အကြောင်းကို general abstraction ကောင်ကောင်းရသွားပါပြီ၊ ထပ်ပြီးလေ့လာချင်သေးတယ်ဆိုရင် DMA Controller တွေရဲ့အလုပ်လုပ်ပုံတွေ၊ data တွေ transfer ဘယ်လိုလုပ်သွားလဲဆိုတဲ့ အသေးစိတ်တွေထပ်လေ့လာလို့ရပါသေးတယ်။

Virtual Memory



ကျနော်အရင် article တွေမှာ memory management အကြောင်းကို part 1 2 ခွဲပြီးရေးခဲ့ဖူးပါတယ်။ အခု virtual memory ဆောင်းပါးမှာ memory management တုန်းက အကြောင်းတွေပြန်ပါတဲ့အတွက် အဲ့ဒီ article

တွေပြန်ဖတ်ဖို့ recommend လုပ်ပါတယ်။ link တွေကို comment မှာတင်ပေးထားပါတယ်။

ဒါဆိုစလိုက်ကြရအောင်။ Computer တစ်လုံးက သူ့မှာ Physically သွင်းထားတဲ့ memory ထက်ပိုပြီးတော့ သုံးနိုင်ပါတယ်။ ဘာလို့လဲဆိုတော့ virtual memory ကြောင့်ပါ။ virtual memory က hard disk ကိုသုံးပြီးတော့ ram ကို emulate လုပ်ပေးပါတယ်။ အသေးစိတ်ပြောရမယ်ဆိုရင် ဥပမာ program တစ်ခု load လုပ်လိုက်ပြီဆိုပါစို့၊ ဒါပေမယ့် run လိုက်တဲ့ program က physical memory RAM ထက်ပိုပြီး memory ကုန်မှာ ၊ ပုံမှန်အတိုင်းဆို virtual memory သာမရှိရင် program က crash ဖြစ်သွားလိမ့်မယ်၊ ဘာလို့ဆို လိုအပ်တဲ့ memory က ရှိနေတဲ့ physical memory(RAM) ထက်ပိုများနေတာကြောင့်။ ဒီနေရာမှာ virtual memory ဆိုတာဝင်လာပြီး Program ရဲ့လိုအပ်တဲ့ memory ကို hard disk သုံးပြီးတော့ ram ကို emulate လုပ်ရင်းထိန်းပေးသွားတယ်။ တစ်ခုတော့ ရှိတာပေါ့၊ RAM ကမလောက်လို့ disk ထဲက virtual memory နဲ့ run နေတဲ့ အချိန်မှာတော့ program ရဲ့performance ကအတော်လေးနှေးပါလိမ့်မယ်။ သိတဲ့အတိုင်းပဲ RAM နဲ့ disk နဲ့ speed က အဆတစ်ထောင်လောက်ကွာပါတယ်။

Virtual memory ကိုသုံးရတဲ့ အဓိက အချက်ကတော့ နှစ်ချက်၊ ပထမတစ်ချက်ကတော့ အခုပြောခဲ့တဲ့အတိုင်း disk ကိုသုံးပြီးတော့ physical memory ကို extend လုပ်ပေးနိုင်တယ်၊ နောက်တစ်ချက်က ကျနော် memory management ဆောင်းပါးတွေရေးတုန်းကပြောခဲ့တယ်။ virtual address တွေကို physical address အဖြစ်ပြောင်းပေးတယ်။ (Memory management unit

လို့ခေါ်တဲ့ကောင်က ပြောင်းပေးတာ)၊ ပြန်နွှေးတဲ့အနေနဲ့ memory management က တစ်ပိုဒ်လောက်ကိုပြန်ထည့်ပေးထားတယ်။

process တစ်ခုက code တွေဆီကနေ reference ယူထားတဲ့ logical address (memory cell, storage element, network host etc.) တွေရှိပါတယ်။ program တစ်ခု memory allocate လုပ်တဲ့အချိန်မှာ OS က logical address တွေကို memory management unit (MMU) ကိုသုံးပြီး physical address အဖြစ် map လုပ်ပါတယ်။ အသေးစိတ်ရှင်းရမယ်ဆို logical address က program တစ်ခု run လိုက်တဲ့အချိန်မှာ CPU က generate လုပ်ပေးထားတာဖြစ်ပြီး physically မရှိဘဲ virtually ပဲရှိပါတယ်။ physical address ကတော့ memory ထဲမှာကို physical location နဲ့ရှိပါတယ်။ MMU ကတစ်ခုနဲ့တစ်ခုကို correspond လုပ်နိုင်ဖို့အတွက် ကြားခံ computation လုပ်ပေးတဲ့အရာတစ်ခုလို့ပြောလို့ရပါတယ်။

အိုကေ ဒါဆို virtual memory ကိုပြန်ဆက်ရအောင်၊ ဒီနေရာမှာ virtual memory နဲ့ physical memory ကိုရောသွားနိုင်ပါတယ်။ ရောစရာမလိုပါဘူး၊ virtual memory က program က သုံးတယ်၊ physical memory RAM ကိုတော့ system ရဲ့ hardware တွေကသုံးပါတယ်။ ဆိုတော့ ပြန်သုံးသပ်ရမယ်ဆိုရင် program ကို run လိုက်ပြီဆို virtual memory ကိုသုံးလိုက်တယ်၊ ပြီးရင် MMU က vm ကို pm အဖြစ်ပြောင်းလိုက်ပြီး system ကို consume လုပ်စေပါတယ်။

virtual memory ကိုတော့ ယေဘုယျအားဖြင့် demand paging technique ကိုသုံးပြီးတော့ ဆောက်ပါတယ်။ demand paging မှာ secondary storage တစ်ခုရှိမယ်၊ disk ပေါ့။ program တစ်ခု run လိုက်လို့ RAM က memory လိုအပ်တဲ့ အချိန်မှသာ disk ကနေယူသုံးတယ်။ swapping technique သုံးတဲ့

paging system နဲ့သွားဆင်ပါတယ်။ (swapping တွေ paging တွေကို memory management article မှာရေးပေးခဲ့ပါတယ်။)

နောက်ထပ်ထပ်ပြီးလေ့လာချင်သေးရင်တော့ virtual memory ကိုတည်ဆောက်တဲ့ demand paging အသေးစိတ်တို့ segmentation အသေးစိတ်တို့ထပ်လေ့လာလို့ရပါသေးတယ်။

ဒီ ဆောင်းပါး series လေးရေးဖြစ်သွားတဲ့ အကြောင်း သို့ ဆောင်းပါးနိဒါန်း

အစကတော့ ကိုယ့်ဘာသာပဲလေ့လာဖို့စဉ်းစားထားပြီးမှ အရှေ့မှာလဲ data structures series တစ်ခုရေးထားတာဆိုတော့ အခုလည်း OS series ရေးရင်ကောင်းမယ်ဆိုပြီး စိတ်ကူးပေါ်ရင်းနဲ့ရေးဖြစ်သွားတာပါ။ OS series ကတော့ DS series လောက်မများပါဘူး။ Operating system ကိုချဲ့ပြီးလေ့လာမယ်ဆိုရင်တော့ topic တော်တော်များပါတယ်။ ဒါပေမယ့် ကျနော်က programming perspective ဘက်ကနေပြီးကြည့်ပြီး programmer တွေအတွက် သိထားလို့ရှိရင် ပိုကောင်းနိုင်မယ့် အပိုင်းတွေကို ထုတ်နုတ်ပြီးရေးလိုက်တာပါ။ ဥပမာ Processes, threading, scheduling အကြောင်း တို့ memory management တို့ စသည်ဖြင့်ပေါ့။

မသိထားရင် program ရေးလို့မရတော့ဘူးလားဆိုတော့ မဟုတ်ပါဘူး၊ ဒါပေမယ့်သိထားလို့ရှိရင် ကိုယ်ရေးရတဲ့ code တွေက behind the scene

မှာဘယ်လိုအလုပ်လုပ်နေတယ်ဆိုတာ အသေးစိတ်ကြီး မဟုတ်ရင်တောင်မှ ယေဘုယျသဘောတရားတွေအနေနဲ့ အော် ငါဒီလိုရေးလိုက်ရင် OS level မှာ ဒီလိုဖြစ်သွားပါလားဆိုတာ သိနေလိမ့်မယ်။ ရှင်းရှင်းပြောရရင် ကိုယ်လုပ်တဲ့အလုပ်အပေါ်မှာပိုပြီး ယုံကြည်မှုရှိလာမယ်ပေါ့ဗျာ။

အခု OS series မှာတော့ အခု conclusion ပါအပါမှ article 10 ခုပဲရှိမယ်ဗျ။ အာမခံပေးနိုင်တာကတော့ အပေါ်မှာပြောခဲ့သလိုပဲ programs တွေ run သွားပြီဆို behind the scene OS level မှာဘယ်လိုဖြစ်သွားတယ်ဆိုတာကို abstraction ကောင်းကောင်းရသွားမယ်လို့ ယုံကြည်ပါတယ်။ memory management အကြောင်းတို့ virtual memory အကြောင်းတို့ကိုလည်း ပိုပြီး သိသွားတာပေါ့။ article တိုင်းမှာ ထပ်ပြီး detail လေ့လာချင်သေးရင် ဘာတွေထပ်လေ့လာလို့ရသေးတယ်ဆိုတာကိုပါ suggestion ပေးထားပါတယ်။ OS ကို detail ထပ်လေ့လာချင်သေးတယ်ဆိုရင် online course တစ်ခုကိုလည်း recommend လုပ်လိုက်ပါတယ်။
(https://www.udacity.com/course/introduction-to-operating-systems--ud923)

See You Later

References/Research လုပ်ထားတာက တော့ medium , geekforgeeks နဲ့ တစ်ခြား website တွေက article တွေကပဲဖြစ်ပါတယ်။