Operating Systems

Isfahan University of Technology
Electrical and Computer Engineering Department

Zeinab Zali

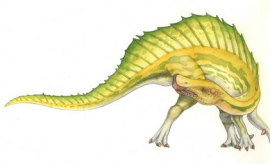
Process Management

Reference: Operating System Concepts book slides

مدیریت فرآیند

Process Concept

- An operating system executes a variety of programs that run as a process.
- **Process** – a program in execution; process execution must progress in sequential fashion. No parallel execution of instructions of a single process
- Program is **passive** entity stored on disk (**executable file**); process is **active**
- Program becomes process when an executable file is loaded into memory
  - Execution of program started via GUI mouse clicks, command line entry of its name, etc.
- One program can be several processes
  - Consider multiple users executing the same program

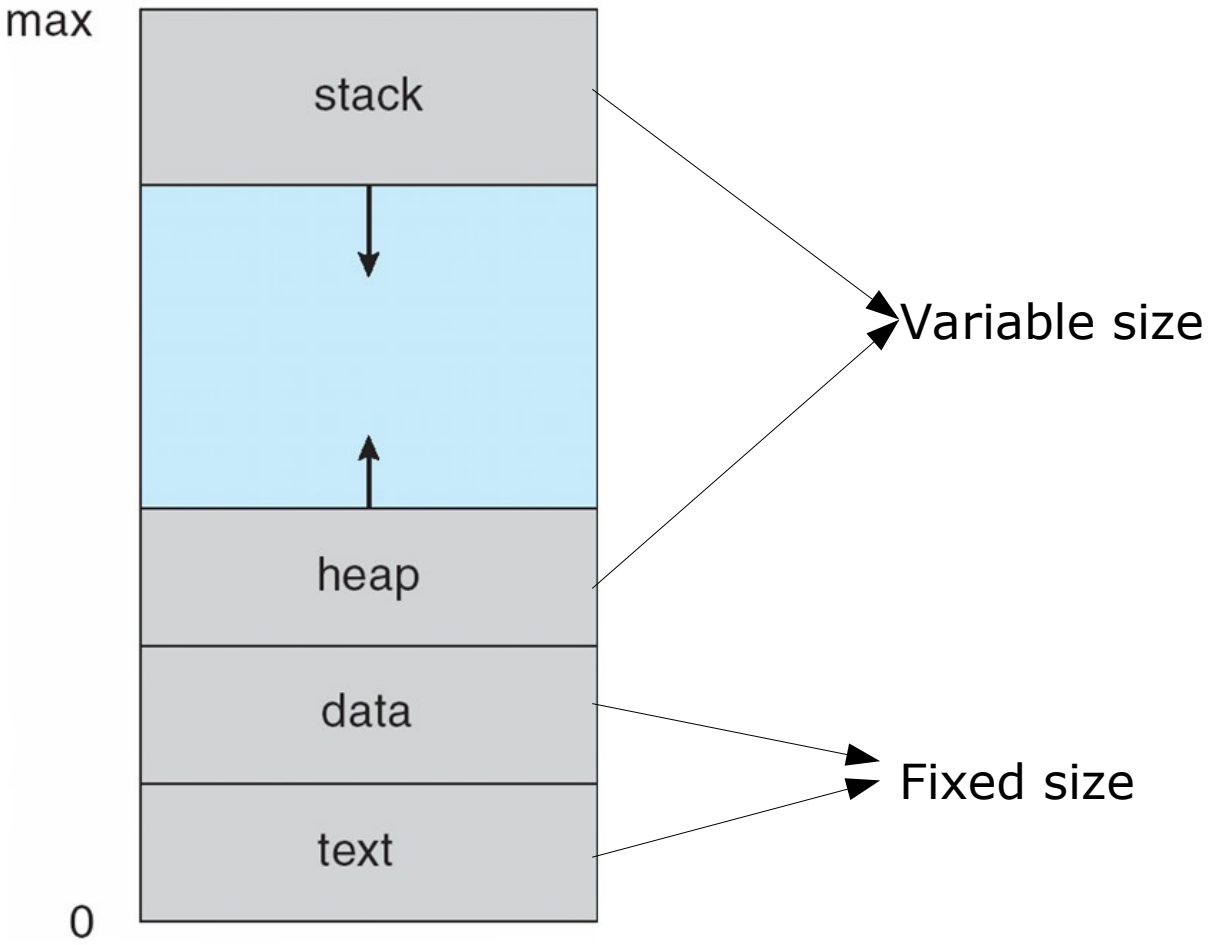


-
یک برنامه وقتی که در حال اجرا است بهش میگیم پروسس
به صورت متدوال پروسس باید خطوطش به ترتیب و دنبال هم توسط سی پی یو اجرا بشه
به عبارت دیگر پروگرم بخش passive برنامه است که روی دیسک ذخیره میشه ولی پروسس بخش active است
وقتی پروگرم تبدیل میشه به پروسس ینی پروسس لود شده توی مموری. چجوری این اتفاق می افته؟
ممکنه برنامه از طریق های مختلف اجرا بشه ممکنه با GUI اجرا بشه با کلیک ماوس یا توی کامند لاین یا ممکنه یک برنامه حین اجرا یک برنامه دیگه رو خودش اجرا بکنه که میشه بحث مالتی پروسس
به هر حال برنامه تبدیل به پروسس میشه و پروسسه که در حال اجرا است
از یک برنامه ممکنه پروسس های مختلفی داشته باشیم
نکته: با دستور top می تونیم لیست پروسس هایی که در حال اجرا هستند رو ببینیم

Process Parts

- The program code, also called **text section**
- Current activity including **program counter**, processor registers
- **Stack** containing temporary data
- Function parameters, return addresses, local variables
- **Data section** containing global variables
- **Heap** containing memory dynamically allocated during run time

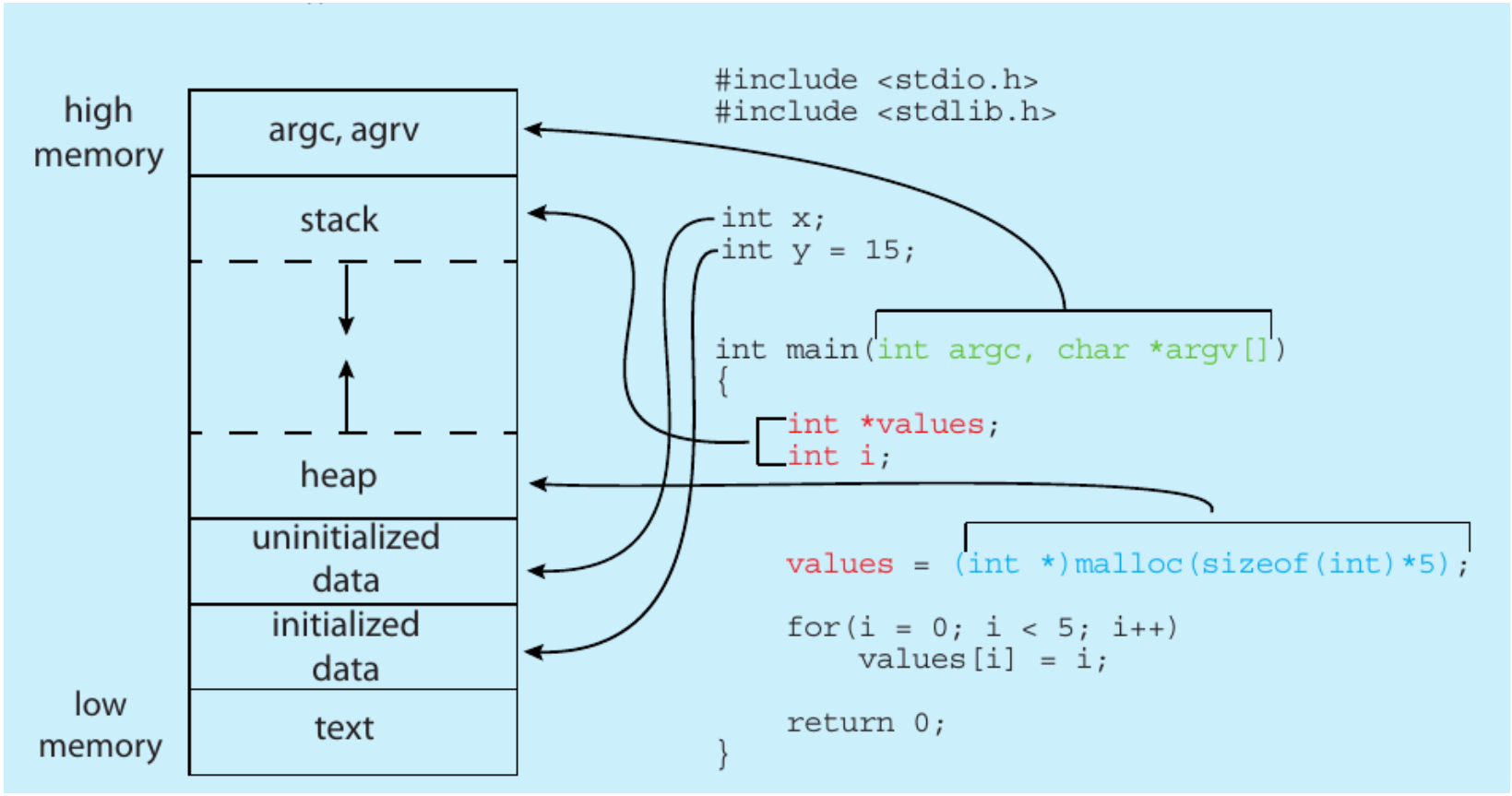
Process in Memory



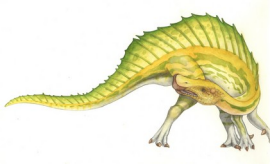
stack and heap sections grow toward one another,
the operating system must ensure they do not overlap one another

-
این شماتیکی از فضایی است که هر پروسس توی حافظه اشغال میکنه
text منظور همون کد برنامه است = code
data مربوط میشه به متغییرهای گلوبالی که توی برنامه تعریفشون کردیم
heap , stack دو تا حافظه دیگه هستند که توی اجرای برنامه به مرور می تونه کم و زیاد بشه
ابیه فضای که خالی است و هنوز اطلاعاتی توش نرفته
پروسس یک فضای ادرسی داره که از ادرس صفر شماره گذاری میشه تا یک ادرس max که این مقدار max چنده وابسته به سیستممون میشه
قسمت دیتا و کد سایزش ثابت است مثالش صفحه بعدی است

Memory Layout of a C Program



بخش text کل این کدمون است

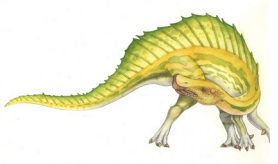


-
heap : جاهایی که حافظه پویا allocate میکنیم مثل وقتی که ازدستور malloc استفاده میکنیم واسه حافظه گرفتن که این از فضای heap گرفته میشه
stack : برای نحوه اجرای خود برنامه استفاده میشه مثلا وقتی که فانکشن ها فراخوانی میشن یکسری ارگومان های ورودی داره که اینا اطلاعاتشون توی استک pass میشن به فانکشن مثلا اگر توی تابع مین یه فانکشن دیگه به اسم f1 فراخوانی کرده باشیم و ورودیش اگر i باشه و وقتی می رسیم به این خط که f1 است i باید بره توی استک و بعد f1 فراخوانی میشه - همینطور اگر مقداری رو ریترن کنیم از f1 این هم توی استک نگهداری میشه - همینطور مقادیری که لوکال هستند واسه فانکشن مثل همین int i , int *values این ها هم توی استک نگهداری میشن

Process State

- As a process executes, it changes **state**
  - **New**: The process is being created
  - **Running**: Instructions are being executed
  - **Waiting**: The process is waiting for some event to occur
  - **Ready**: The process is waiting to be assigned to a processor
  - **Terminated**: The process has finished execution

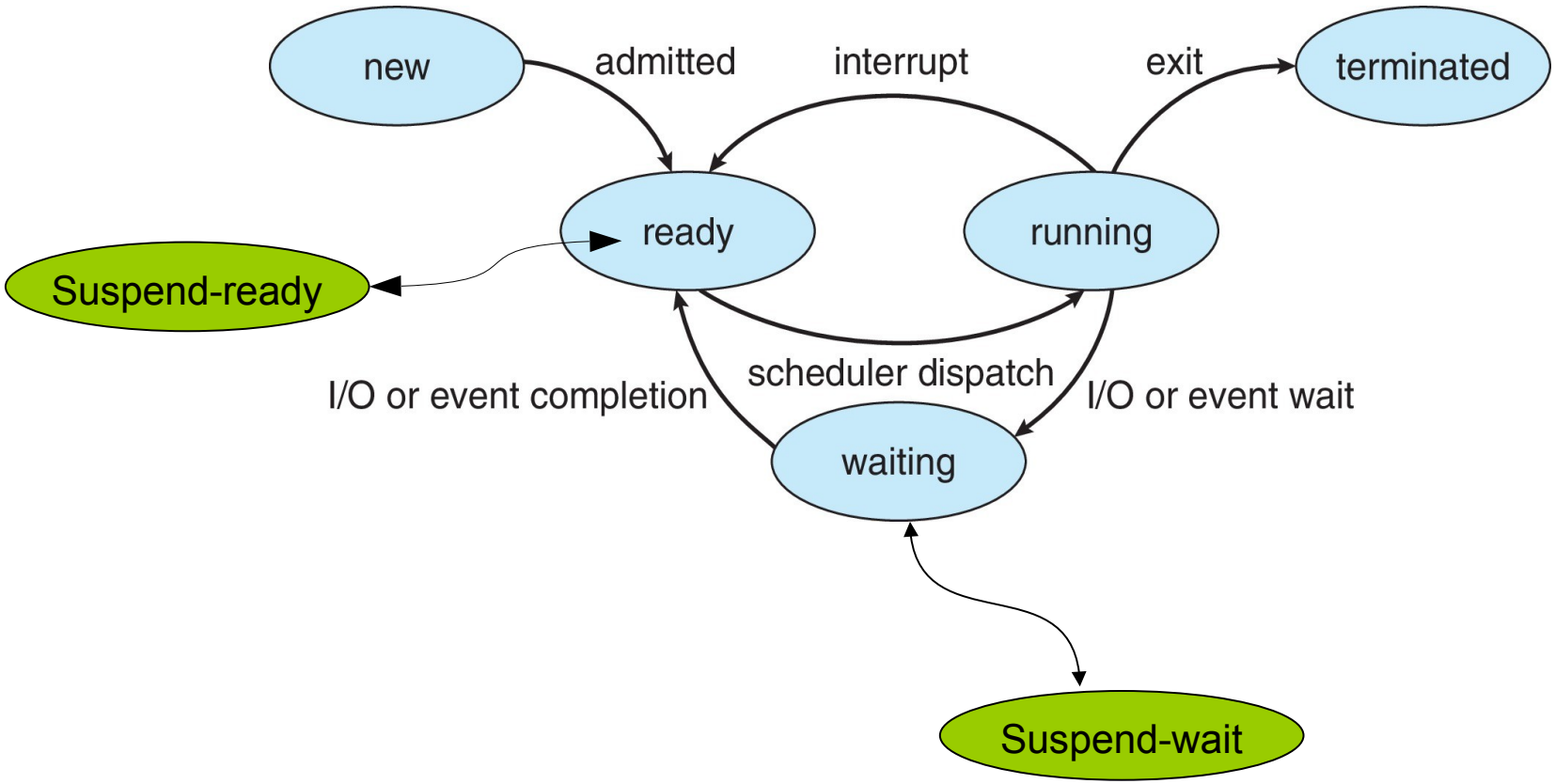
With single CPU, only One program is running
while many programs may be ready or waiting



-
یکسری اطلاعاتی که باید نگهداری کنیم واسه پروسس یکیش state پروسس است
پروسس توی سیستم توی زمان های مختلف توی state های مختلفی قرار داره
new : ینی پروسس تازه ساخته شده ینی لود شده توی مموری ولی سی پی یو هنوز نرفته سراغش که اجراش بکنه
running : اگر سی پی یو در حال اجرای برنامه باشه میگیم برنامه در استیت running است
waiting : اگر برنامه به یک خطی برسه که نیاز داشته باشه یک صبر کنه که یکسری اتفاقاتی بیوفته و بعد اجراشو ادامه بده مثلا میخواد یه سری چیزا از ورودی بخونه و هنوز کامل خونده نشده و.. در حالت کلی دیگه نمیخواد اجرا بشه تا یک اتفاقی بیوفته و بعد بتونه دوباره اجرا بشه
ready : ینی نه منتظر چیزیه و نه در حال اجراست ینی توی این حالت سیستم عامل هنوز بهش سی پی یو نداده و فعلا امادس برای اجرا توی حافظه ولی هنوز instruction هاش توسط سی پی یود در حال اجرا نیستند
terminated : توی این حالت برنامه تموم شده و دیگه در حال اجرا نیست

نکته: اگر یک سی پی یو رو در نظر بگیریم همیشه فقط یک برنامه هست که در حال اجرا است و بقیه برنامه ها یا توی حالت ready هستند یا توی حالت waiting هستند

Diagram of Process State

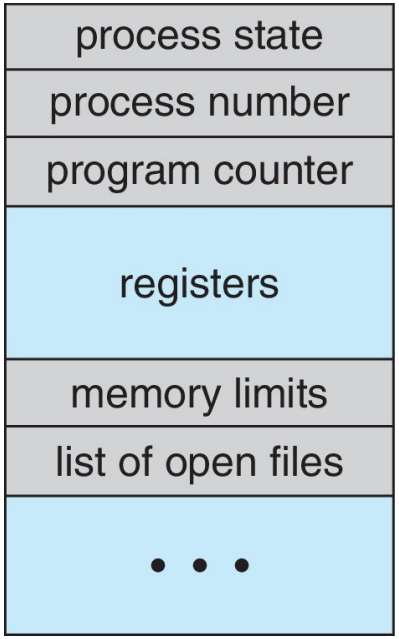


-
شکلی از تغییر حالت برنامه:
admitted : برنامه وقتی برای اولین بار وارد سیستم میشه ینی لود میشه توی مموری میگیم admitted اش کردیم و این دیگه می ره توی حالت ready و می تونه اجرا بشه و اگر سیستم عامل انتخابش بکنه که توی سی پی یو اجرا بشه ینی از ready بره توی استیت running که میشه scheduler dispatch حالا که داره اجرا میشه ولی یه جایی به یه خطی می رسه که یک I/O یا یک event اتفاقی بیوفته در این حالت می ره توی waiting (در واقع این کارهارو سیستم عامل داره انجام میده) حالا اگر اون I/O یا اون event اتفاق افتاد می تونه دوباره بره توی حالت ready ینی توی حالتی که می تونه دوباره انتخاب بشه
یه موقعی هم هست که یک برنامه داره اجرا میشه و نیاز به I/O or event هم نداره ولی مهلت زمانیش تموم شده در این حالت بخاطر interrupt زمان که این interrupt که نوشته تایمر است ینی با این تایمر سیستم عامل متوجه میشه که این مهلت زمانیش تموم شده و می برتش توی حالت ready
suspend چیه؟ ممکنه ما یک برنامه ای رو بخاطر اینکه مموریمون پر شده از مموری منتقلش کنیم به دیسک ینی ممکنه برنامه با اینکه در حال اجرا است ینی هنوز terminated نشده ولی ما پروسس رو می بریمش توی دیسک
نکته: اونایی که هنوز در حال اجرا نیستند رو suspend میکنیم ینی یا توی حالت ready است یا حالت waiting
و بعد اگر جا داشت مموریمون می تونه از دیسک منتقل بشه به مموری باز

Process Control Block (PCB)

Information associated with each process(also called **task control block**)

- Process state – running, waiting, etc.
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files

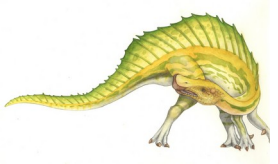


All processor designs include a register or set of registers, often known as the **program status word** (PSW), that contains status information.

-
برای اینکه سیستم عامل بتونه پروسس های مختلف رو همزمان مدیریت کنه و بتونه هر لحظه یکیشونو بده به cpu که اجرا بشن باید یک دیتا استراکچری از اطلاعات پروسس همیشه نگه داره
پروسس اون برنامه ای که لود شده توی مموری پس برنامه هایی که لود شده توی مموری رو واسشون PCB نگه می داریم
بهش task control block هم میگن ولی به PCB مشهور است
چه اطلاعاتی توش است؟ process state و - program counter ینی ما کدوم یکی از Instruction های برنامه رو الان داریم اجرا میکنیم و - cpu register ها ینی وقتی که یک برنامه توی سی پی یو در حال اجرا است سی پی یو یک سری رجیسترهایی داره مثلا رجیسترهای جمع شونده و ... پس رجیسترهای مختلف وجود داره که امکان اینکه ما راحت بتونیم Instruction هارو اجرا کنیم داشته باشیم
بعضی وقتا از لفظ program status word استفاده میکنن که مجموع رجیسترهایی که پروسسور داره برای اجرای یک پروسس را بهش PSW میگن
ادامه بقیه اطلاعات توی PCB : اطلاعات مربوط به scheduling ینی زمان بند سیستم که میخواد تصمیم بگیره که میخواد چه پروسسی رو در لحظه فعلی اجرا کنه -- یکسری اطلاعات راجع به memory management داره ینی این پروسس مدنظر ما کجای حافظه قرار داره -- یکسری اطلاعات accounting داریم ینی مثلا چقدر تا الان از cpu استفاده کرده یا چند کلاک یا چقدر از اجراش مونده و.. -- و در اخر هم یکسری اطلاعاتی که مربوط به I/O هم هست رو داریم ینی مثلا الان براش چه I/O اختصاص داده شده و ...

Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB
- Explore in detail in Chapter 4



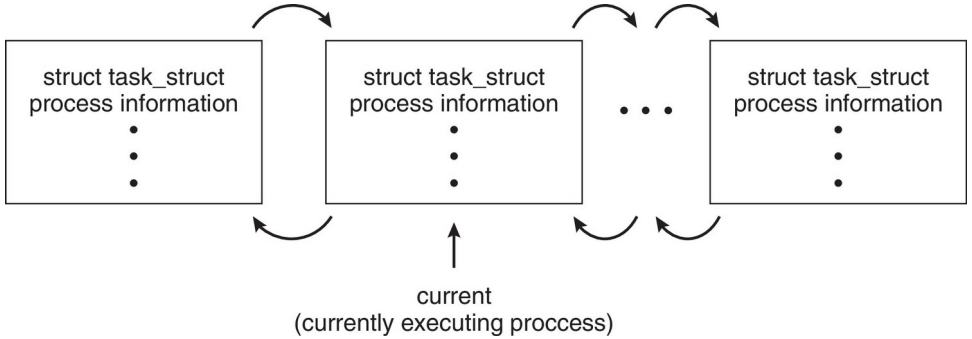
-
: threads
تا الان حرفایی که زدیم واسه یک پروسس سینگل Thread بود اگر پروسس ما بیشتر از یک Thread داشته باشه باید program counter های بیشتری داشته باشیم ینی به ازای تمام Thread ها باید program counter داشته باشیم چون از جاهای مختلفی برنامه میتونه اجرا بشه بنابراین توی PCB ما یه دونه counter نداریم تعداد بیشتری counter داریم

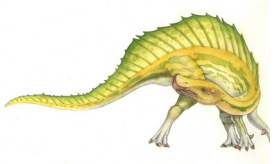
Process Representation in Linux

Represented by the C structure `task_struct`

<include/linux/sched.h>

```
pid t_pid;          /* process identifier */
long state;             /* state of the process */
unsigned int time_slice   /* scheduling information */
struct task_struct *parent;/* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files;/* list of open files */
struct mm_struct *mm;  /* address space of this process */
```





-
توی لینوکس یک استراکچری داریم به اسم task_struct که داره PCB رو نگهداری میکنه

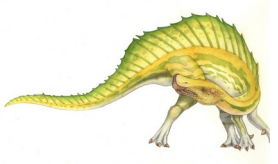
Look inside sched.h and find the corresponding data structures to this chapter

<linux/sched.h>

rb_node
Rq (running queue)
State (runnable, unrunnable, stopped)

Process Scheduling

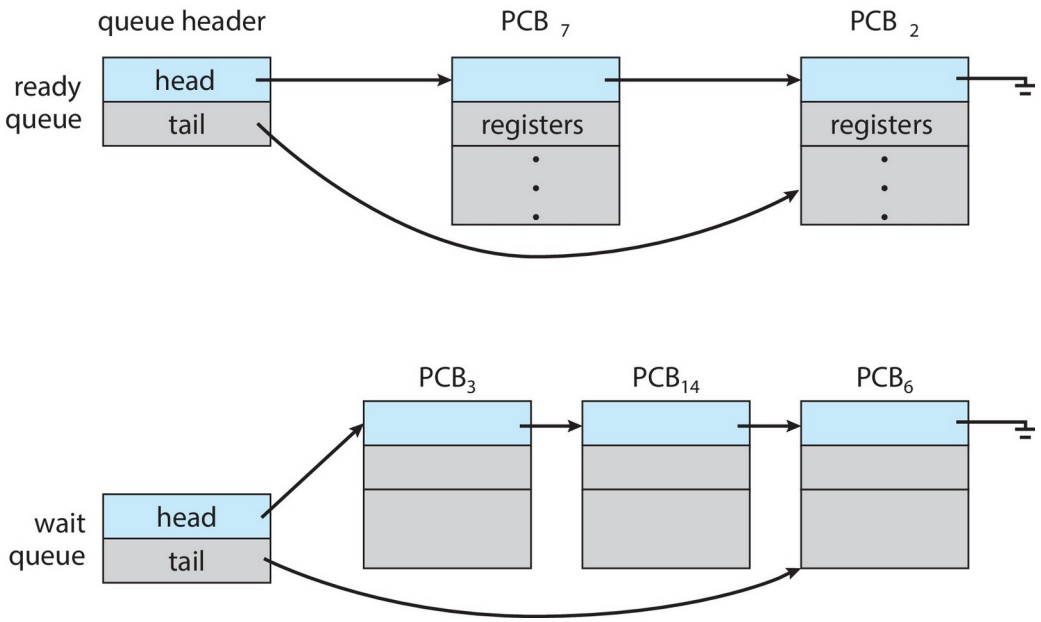
- **Process scheduler** selects among available processes for next execution on CPU core
- Goal -- Maximize CPU use, quickly switch processes onto CPU core
- Maintains **scheduling queues** of processes
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Wait queues** – set of processes waiting for an event (i.e., I/O)
  - Processes migrate among the various queues



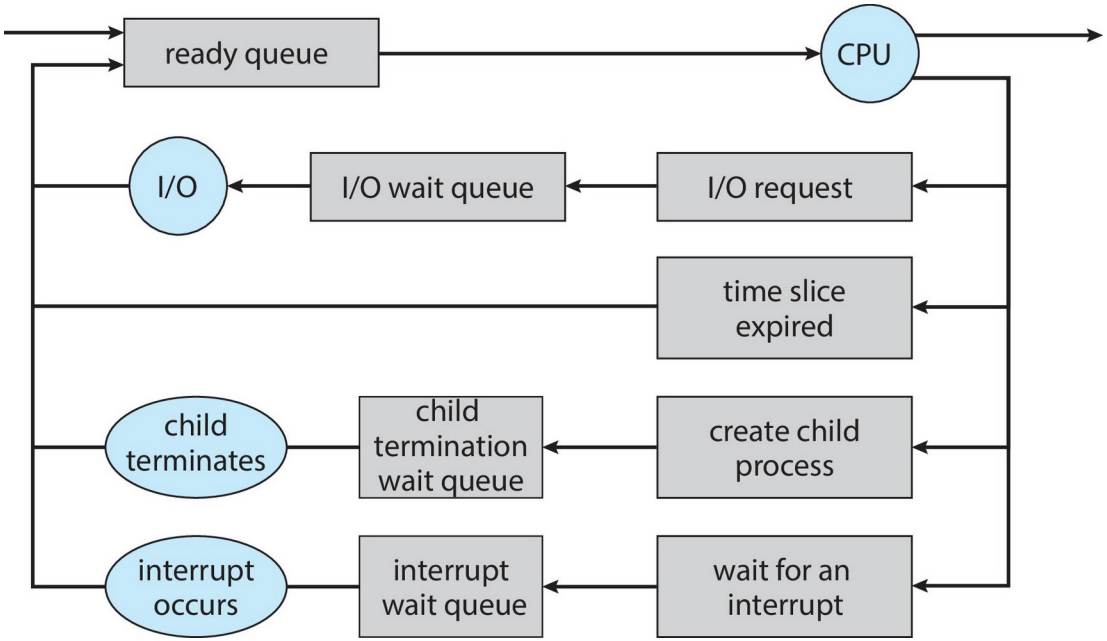
-
scheduler چی کار میکنه؟ نگاه میکنه و براساس یه سری الگوریتم هایی تصمیم میگیره که کدوم یکی از این پروسس هارو توی لحظه بعدی برای اجرا انتخاب بکنه
هدف اینه که ما بیشترین استفاده رو از cpu داشته باشیم و خیلی سریع بتونیم بین پروسس های مختلف سوییچ کنیم
این استراکچری که گفتیم رو سیستم عامل می تونه توی صف های مختلفی نگهداری کنه که بتونه مدیریت اسون تر و بهتر داشته باشه از جمله این صف ها :
ready queue : لیست پروسس هایی که الان ادامه اند و اینارو می تونه توی یک صفی نگه داره
wait queue : و اونایی که منتظر I/O or event هستند رو توی یک صف دیگه نگه داره

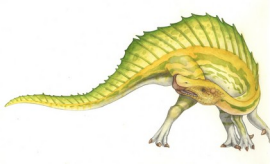
Ready and Wait Queues

این فلاش ها پوینتر هستند



Representation of Process Scheduling

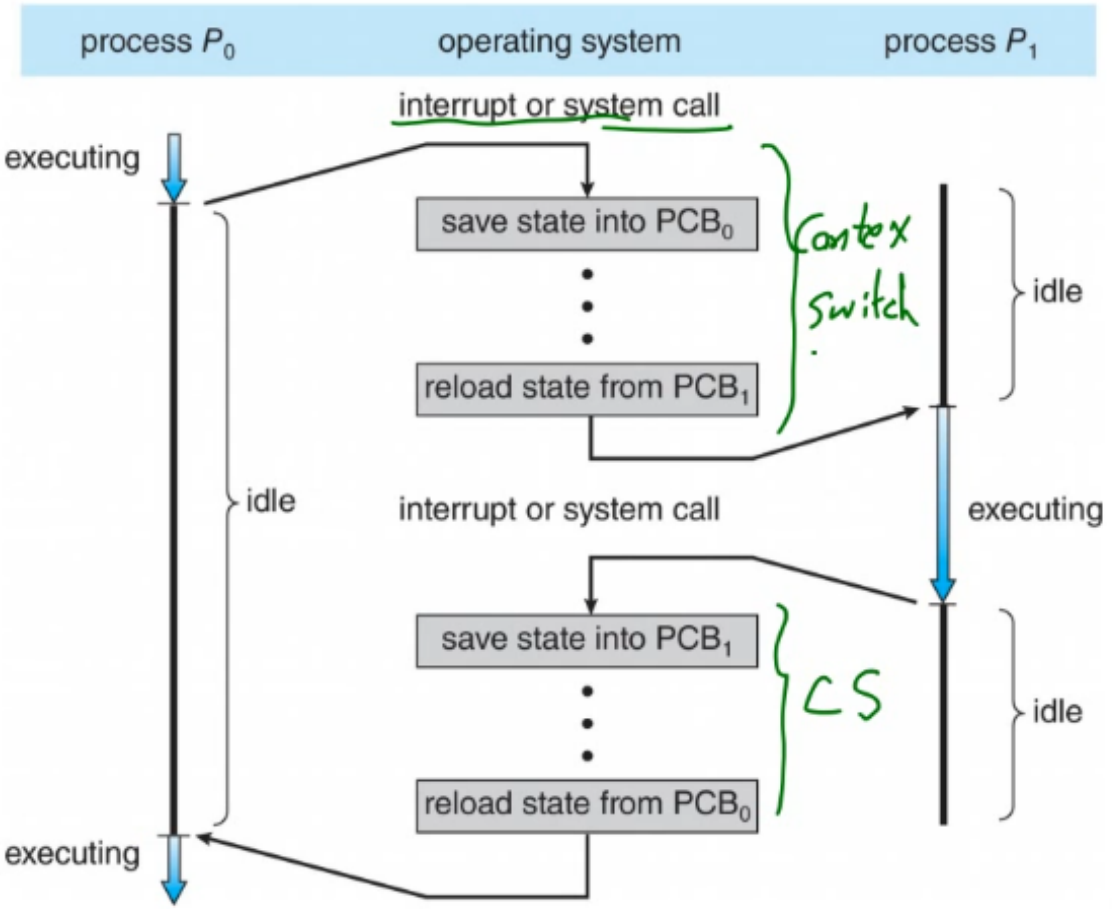




-
این هم یک نمایی از اتفاقاتی است که می تونه برای یک پروسس بیوفته
وقتی یک پروسس از صف ready ها انتخاب میشه که توی cpu اجرا بشه ممکنه حین اجرای Instruction هاش یک اتفاقاتی بیوفته مثلا به خطی برسیم که یک I/O اتفاق بیوفته در این صورت این برنامه اگر I/O نیاز داشته باشه سیستم عامل می فرستش توی صف wait ها و اگر I/O اتفاق افتاد اون وقت می تونه از صف I/O بیاد بیرون و بره توی صف ready ها که یه موقعی scheduler انتخابش بکنه و بیاد توی cpu
ولی ممکنه یه موقع هم یه برنامه I/O نیاز نداشته باشه ولی time slice به پایان رسیده باشه در اینصورت هم باز می ره توی صف ready ها
یا ممکنه یک پروسس فرزند ایجاد کرده و توی حالتیه که باید منتظر اجرای پروسس فرزندش باشه و وقتی تموم شد میتونه دوباره اجراشو از سر بگیره
یا ممکنه یک وقفه دیگه غیر از I/O نیاز داشته باشه که اتفاق بیوفته و باز می ره توی صف که اون اتفاق بیوفته و وقتی که تموم شد می ره توی صف ready ها تا دوباره scheduler اونو انتخاب بکنه و بره توی cpu

CPU Switch From Process to Process

A **context switch** occurs when the CPU switches from one process to another.



-
وقتی که سی پی یو سوییچ میکنه از اجرای یک پروسس به یک پروسس دیگه و عملیاتی که نیازه اتفاق بیوفته که این تغییر پروسس توی سی پی یو اتفاق بیوفته بهش میگیم context switch پس به غیر از اینکه اسکجولر انتخاب میکنه چه پروسسی باید انتخاب بشه بعد از انتخاب اسکجولر یک ماژول دیگه که باید عملیات context switch انجام میده باید اجرا میشه تا کارهایی که نیازه که این پروسس جدید بتونه انجام بده رو اجرا بشه
شکل:
سی پی یو در حال اجرای پروسس p0 است و p1 میخوایم بعد از p0 اجرا کنیم حالا ممکنه یک وقفه ای اتفاق بیوفته الان اینجا میشه همون حالت interrupt or system call و میگه دیگه p0 اجرا نشه و به جاش p1 اجرا بشه پس عملیات context switch انجام میشه
مهمترین بخشی که توی context switch انجام میشه اینه که مثلا اگر p0 در حال اجرا نباشه باید یه سری اطلاعاتی رو توی PCB اش اپدیت کنیم مثلا رجیسترهای سی پی یو که در حال اجرای P0بودند باید توی PCB نگهداری بشن ینی مقدارشون سیو میشه توی PCB0 و استیتشون عوض میشه یا هر اطلاعات دیگه ای که باید اپدیت بشه چون دیگه نمی خوایم الان با P0 کار کنیم و توی این فاصله ای که داره این کارا انجام میشه در واقع سی پی یو نه P0 داره اجرا میکنه و نه P1 واسه همین idleمی بینیم
idle منظور این نیست که سی پی یو هیچ کاری نمیکنه داره قسمتایی که مربوط به operating system هست رو اجرا میکنه
دوست داریم این بخش هایی idle کم باشه و بیشترین زمان سی پی یو روی اجرای پروسس ها باشه
بعد از اون چون P1 میخوایم اجرا کنیم یه سری اطلاعاتی که نیازه از PCB1 لود میشه توی رجیسترهای سی پی یو تا P1 بتونه اجرا بشه
executing ینی P1 داره توی اون قسمت انجام میشه بعد اگر اجرای P1 تموم شد و خواستیم P0 رو اجرا کنیم دوباره یک context switch دیگه باید انجام بشه تا P0 بتونه اجرا بشه
P0, P1پروسس های توی سی پی یو هستند ولی هر کدومشون یه زمان هایی idle هستند حالا ممکنه اینجا اتفاقی که افتاده بریم سراغ P1 این بوده که P0 نیازه به یک وقفه ای داشته یا ممکنه هر چیز دیگه ای باشه مثلا مهلت زمانیش تموم شده یا ...
پس کل این شکل داشت نشون می داد که عملیاتی که نیازه که یک پروسس تغییر بکنه توی سی پی یو

Context Switch

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is pure overhead; the system does no useful work while switching
  - The more complex the OS and the PCB ➔ the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once

What is the difference between Interrupt handling and context switch?

-
منظور از Context همون اطلاعاتی است که توی PCB نگهداری میکنیم
بعضی از سخت افزارها امکان اینو دارند که بیشتر از یک دسته رجیستر داشته باشند و این به چه دردی میخوره ؟ این باعث میشه که ما در ان واحد بیشتر از یک PCB رو بتونیم اطلاعاتشو لود کنیم توی سی پی یو و این یکم زمان رو برای ما سیو میکنه

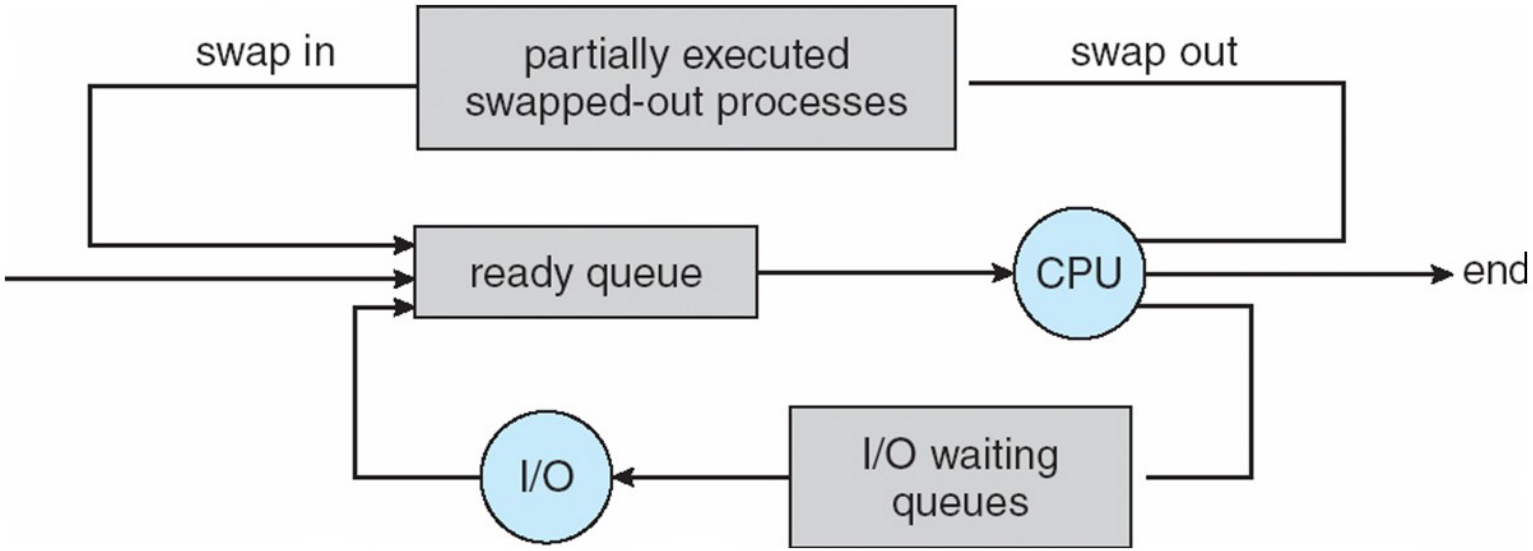
Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Load balancing of I/O and CPU bound processes
  - Long-term scheduler strives for good ***process mix***

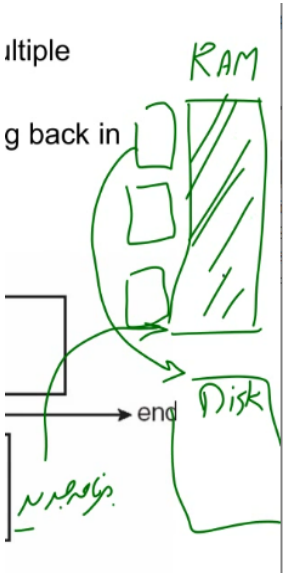
-
Schedulers های سیستم رو به چند دسته تقسیم می کنند:
Short-term scheduler : همون Schedulers که همیشه مد نظر ما است چرا بهش Short-term میگن چون frequently این اتفاق می افته ینی در فاصله زمانی های خیلی کوچیک کوچیک این اتفاق می افته چون میخوایم کاربر متوجه نشه که سیستم در هر لحظه داره یکی از این پروسس هارو اجرا میکنه و سریع بین این ها میخوایم سوییچ کنیم
Long-term scheduler : اون اسکجولری که انتخاب میکنه کدوم پروسس بیاد توی صف readyها ینی با درخواست انی کاربر نخوایم برنامه هارو اجرا کنیم
این حالت درجه multiprogramming سیستم را مشخص میکنه. درجه multiprogramming سیستم ینی چی؟ ینی تعداد پروسس های موجود در سیستم ینی نه اونایی که حتما در حال اجران ینی اونایی که دارای PCB در لیست های task-struct هستند
I/O-bound process : به پروسسی که بیشترین لحظات رو داره I/O انجام میده مثلا زمان های زیادی باید یه یکسری اطلاعاتی رو از توی دیسک بخونه
CPU-bound process : اونایی که بیشتر دارند از cpu استفاده میکنند و خیلی به مراجعه به دیسک نیاز ندارند . ادامه ص بعدی ...

Addition of Medium Term Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



-
ادامه..
نکته: اگر سیستم ما بیشتر برنامه های CPU-bound process داشته باشه زیاد خوب نیست یا اگر همش I/O-bound process باشه چون اگر CPU-bound باشه مدام داره از cpu استفاده میکنه مگر اینکه ما خودمون ازش cpu رو بگیریم و اگر I/O-bound باشه توی لحظه هایی که می ره از I/O استفاده بکنه ما می تونیم اون برنامه رو از cpu بگیریم و یک برنامه دیگه رو به cpu بدیم پس اگر ما ترکیب خوبی از برنامه های CPU-bound و I/O-bound در سیستم در حال اجرا داشته باشیم زمان هایی که برنامه I/O-bound می ره منتظر I/O باشه ما می تونیم اون برنامه CPU-bound بدیم به سیستم که اجرا بشه
پس یکی از کارهای Long-term scheduler توی سیستم هایی که این قابلیت رو دارند یک ترکیب خوبی از برنامه های I/O-bound و CPU-bound رو بتونن توی سیستم لود کنن
از طرفی سیستم می تونه توی حالت suspend بره که کی این کارو انجام میده؟ یک اسکجولری به اسم Medium-term چرا این کارو میکنه؟ ممکنه رم فیزیک سیستم کاملا پر شده باشه از اطلاعات کل پروسس ها و یک پروسس دیگه رو کاربر می خواد بازش کنه و سیستم نمیخواد به کاربر بگه که من دیگه پر شدم و نمیتونم اینو اجراش کنم پس کاری که میکنه اینه که توی رم نگاه میکنه و می بینه کدوم یکی از این پروسس ها است که در حال اجرا نیست و اونو برمیداره و می بره توی دیسک نگهداری میکنه و الان امکان که برنامه جدید رو بیاره توی رم داره
به اون فضایی که توی دیسک برای این کار استفاده میشه هم swapping میگن



Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

ps -el

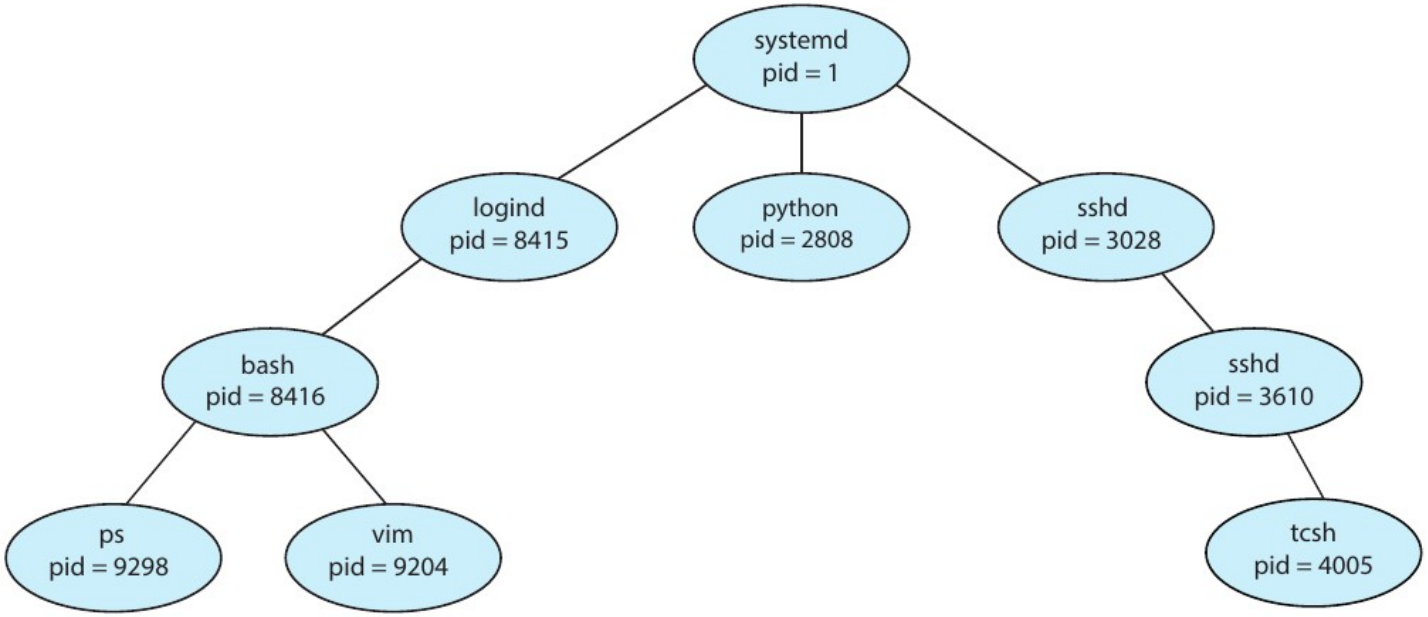
pstree

Prevent overloading the system by too many child processes

-
توی سیستم عامل های شبیه لینوکس نحوه ایجاد پروسس ها به صورت والد و فرزند است ینی هر پروسسی که داریم می تونه پروسس های دیگه ای ایجاد بکنه و اون پروسس های ایجاد میشه می شن پروسس فرزند اون پروسس اولیه و به اون پروسس اولیه والد میگیم
ممکنه ساختار درختی برای پروسس ها داشته باشیم
برای هر یک از این پروسس ها برای اینکه شناسایی بشن توی سیستم به صورت یکتا یک pid وجود داره
قواعدی که داریم اینجا:
یکی از قواعد resource sharing است بین پروسس فرزند و والد و میتونه حالتی پیش بیاد که تمام resources ها شیر باشه بین والد و فرزند یا اینکه فرزند یک زیر مجموعه ای از resources های پدر را شیر کرده باشه یا اینکه اصلا هیچی شیر نکرده باشند
اینکه چی شیر میکنند یا چی شیر نمیکنند و چیا شیر هست یا نیست این توی نحوه ایجاد کردن اون پروسس فرزند میشه اپشن هایی رو در نظر گرفت که با حالت های مختلف بشه ایجادش کرد
اپشن هایی که برای execution داریم:
اگر والد در حال اجرا بوده و بعد یه جایی از این خط والد میاد پروسس فرزندشو ایجاد میکنه حالا اون خطی که پروسس فرزند رو ایجاد کرد می تونه متنظر باشه و متوقف بشه که پروسس فرزند کامل اجرا بشه و بعد دوباره خودش اجرا بشه یا نه ممکنه همزمان این دوتا اجرا بشن ینی این فرزندشو ایجاد بکنه و فرزند شروع بکنه به اجرا شدن و از طرفی خودش هم در حال اجرا کردن باشه
اگر بخوایم لیست پروسس های سیستم رو ببینیم از دستور ps می تونیم استفاده بکنیم

A Tree of Processes in Linux

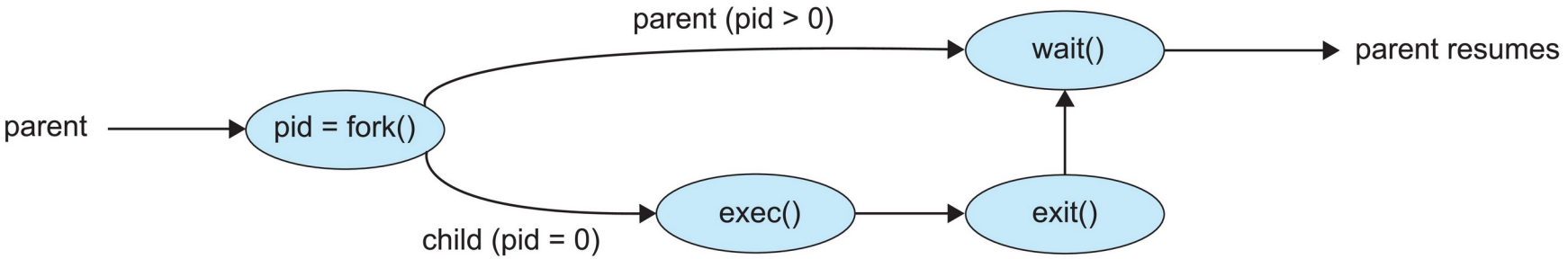
pstree

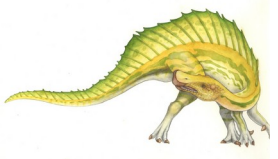


-
چرا ساختار والد و فرزندی در نظر گرفتن؟ اینجا یک مدیریت کنترل شده ای وجود داره روی ساخت پروسس ها و باعث میشه که یک دفعه سیستم overload نشه و ما توسط والد هر پروسسی امکان کنترل اون پروسس رو داشته باشیم (این تیکه ابی صفحه قبل بود)

Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - `fork()` system call creates new process
  - `exec()` system call used after a `fork()` to replace the process' memory space with a new program
  - Parent process calls `wait()` waiting for the child to terminate

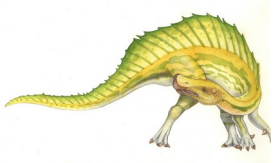




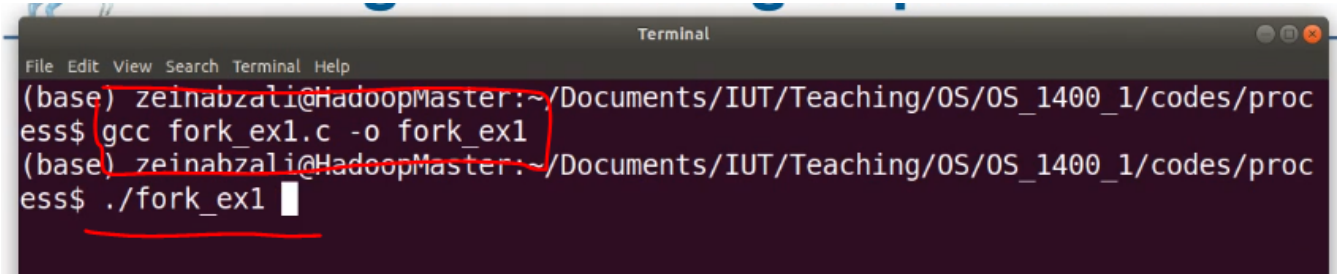
-
اولین کاری که باید انجام بدیم و اولین API که باید داشته باشیم برای ساخت پروسس است
دستور fork یک دستوری است که پروسس های جدید رو ایجاد میکنه
در واقع وقتی که fork انجام میشه ینی یک پروسس والدی داریم که fork براش انجام شده و از اونجا به بعد دو تا شاخه داریم یکیش برای والد است و یکیش برای فرزند
وقتی که fork فراخوانی میشه یک پروسس شبیه والد ایجاد میشه
exec این میتونه یک پروسسی رو یا یک برنامه ای رو که توی سیستم داریم به عنوان پروسس توی فضای این فرزند لودش بکنه پس exec توی هر برنامه ای که فراخوانی بشه یک پارامتر اصلی که باید بگیره یک برنامه باینری دیگه است و وقتی فراخوانی بشه اون برنامه باینری رو شروع میکنه به اجرا کردن
حالا اگر توی این فرزند دستور exec داشته باشیم به محض اینکه این exec اجرا میشه اون پروسسی یا برنامه ای که به عنوان ورودی اسمشو اینجا به exec دادیم به عنوان پروسس لود میشه توی همین فضای ادرس
wait توی والد فراخوانی شده و exit توی فرزند
wait وقتی فراخوانی میشه توی یک والد منتظر می مونه که فرزندش تموم بشه و فرزند هم وقتی exit رو فراخوانی میکنه میتونه توسط exit یک پارامتری رو به عنوان ریترن برای اینکه به والدش اعلام بکنه چجوری تموم شده بهش برگردونه و این پارامتر توی ارگومان های wait دریافت میشه
نکته: اگر والد wait را فراخوانی نکرده باشه فرزند و والد میتونن همزمان اجراشونو ادامه بدن

C Program Forking Separate Process

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```



-
مثال از فراخوانی fork
خروجی fork از نوع pid_t که یک شناسه پروسس به ما میده و یک عدد integer است
اگر مقداری که fork برمیگردونه بیشتر از صفر باشه ما می فهمیم که الان توی کد والد هستیم و این مقدار pid میشه pid فرزند تولید شده
و اگر مقدار pid صفر باشه یعنی الان توی کد فرزند هستیم
نکته: تا قبل از اینکه exec فراخوانی بشه ما یک کد داریم برای والد و فرزند
اگر pid کمتر از صفر باشه یعنی اصلا نتونسته fork رو انجام بده
نکته: هیچ وقت exec خالی نداریم همیشه exec بعدش یه سری چیز میاد
برای execlp باید اسم اون برنامه رو بدیم که اینجا میشه ls و مسیر رو بهش میدیم که کدوم ls رو میخوایم اجرا بکنیم که میگه اون ls که توی /bin است و اون NULL هم که هست برای اینکه که اگر ls ارگومان ورودی می خواست اونجا بنویسیم که فعلا این ls ارگومان ورودی نمیخواست واسه همین NULL دادیم
حالا میخوایم این برنامه رو توی لینوکس اجرا بکنیم:



Practice

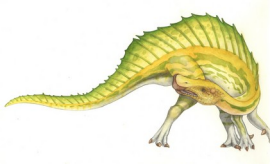
What is the output of following code? How many processes are created?

```
Main(){
   pid_t pid1, pid2, pid3;
   pid1 = fork();
   wait()
   pid2 = fork();
   if (pid1 == 0 or pid2==0){
       printf('new child process');
    }
    pid3 = fork();
    printf("End of the process");
}
```

تمرین
خروجی کد زیر چیست؟ چند فرآیند ایجاد می شود؟

Process Termination

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - Returns status data from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **kill `()`** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates



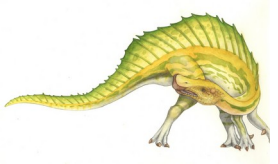
-
هر پروسسی وقتی که exit را فراخوانی میکنه تموم میشه و با استفاده از status که میتونه به عنوان ارگومان بگیره میتونه وضعیتش رو به پروسس والدش اطلاع بده و این وضعیت توسط تابع wait دریافت میشه
و وقتی که exit شد دیگه resources های که مربوط به اون پروسس بوده می تونه برگرده به سیستم عامل
kill اگر والدی تحت یک شرایط خاص به صورت دستی خودش پروسس فرزنداشو تموم کنه ینی فرزندانش به exit نرسیدن یا اینکه کلا کدشون تموم نشده ولی والد می خواد اونارو ببنده چرا؟
- ممکنه فرزند resources های زیادی allocated کرده و والد بخواد جلوگیری بکنه از این کار
- یا مثلا کاری که بهش assigned کرده تموم شده مثلا میخواسته به یک نتیجه ای برسه مثلا می خواسته یه سری محاسبات انجام بده که جواب به یه حد خوبی برسه اگر رسید دیگه ادامه نمیده ینی دیگه لزومی نداره پروسس فرزند ادامه دیگه پیدا بکنه و پروسس والد می تونه اونو kill اش بکنه
- ممکن است توی یک سیستمی فقط در صورتی که والد زنده هست اجازه بدیم فرزند هم باشه پس وقتی والد کارش تموم شد دیگه نیازی به اون فرزند نباشه پس والد میتونه فرزند رو ببنده و خودش هم تمام بشه

Process Termination

- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the `wait()` system call. The call returns status information and the pid of the terminated process

```
pid = wait(&status);
```

- A process that has terminated, but whose parent has not yet called wait() , is known as a zombie process
- if a parent did not invoke wait() and instead terminated, thereby leaving its child processes as orphans



-
توی بعضی از سیستم ها به طور اتوماتیک به این صورته که وقتی که والد تموم شده فرزندانش هم به صورت یکی یکی پشت سر هم بسته میشن که بهش cascading termination میگیم : توی لینوکس به صورت پیش فرض اینطوری نیست ینی اگر والد تموم بشه فرزند می تونه به اجراش کماکان ادامه بده
wait : والد می تونه با استفاده از wait برای تمام شدن اجرای یک فرزندی صبر بکنه و wait یک خروجی هم داره که این خروجی pid فرزندیه که تمام شده
zombie process : اگر یک پروسسی تمام بشه و والدش wait را فراخوانی نکرده باشه به اون پروسس میگیم zombie process چرا؟ چون pcb وقتی که wait فراخوانی بشه از بین می ره و اگر wait فراخوانی نشه pcb اون پروسس تا یه مدتی توی سیستم می مونه
orphans : اگر یک والدی wait را فراخوانی نکرده باشه و تمام بشه و فرزندش هنوز زنده باشه ینی اجرای فرزند هنوز تمام نشده دز اینصورت چون اون پروسس فرزند دیگه والدی نداره بهش orphans یا یتیم میگیم ولی خوبیش اینه که سیستم عامل میاد فرزندی اونو قبول میکنه

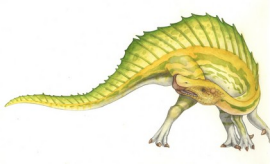
Process Termination

- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system

All processes transition to zombie state when they terminate, but generally they exist as zombies only briefly.
Once the parent calls wait(), the process identifier of the zombie process and its entry in the process table are released.

```
pid = wait(&status);
```

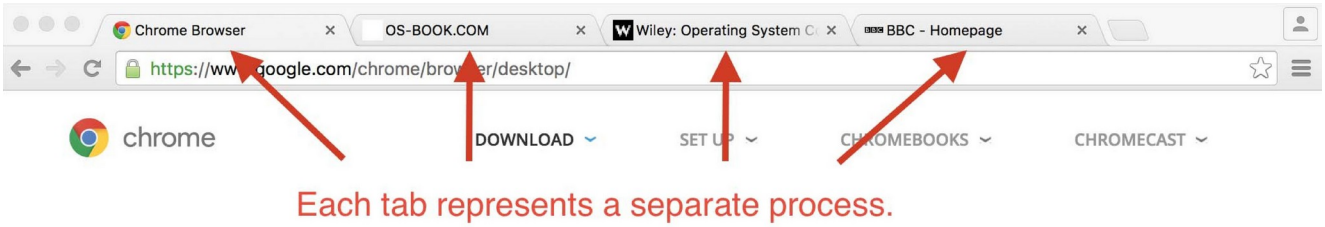
- A process that has terminated, but whose parent has not yet called wait() , is known as a zombie process
- if a parent did not invoke wait() and instead terminated, thereby leaving its child processes as orphans



-
همه پروسس های سیستم یک مدت محدودی زامبی هستند تا اینکه والدشون wait را فراخوانی بکنه و وقتی فراخوانی کرد دیگه از اون حالت زامبی خارج میشن و pcb شون ازاد میشه ولی اگر والدی کلا wait را فراخوانی نکنه اون موقع ممکنه این زامبی بمونه

Multiprocess Architecture – Chrome Browser

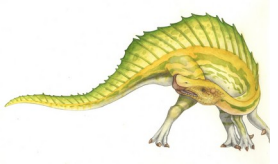
- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    - Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in



-
اینجا یک اپلیکیشن Multiprocess رو می بینیم
یک برنامه Multiprocess پروسس های جدیدی که ایجاد میکنه جدا از خودش هستند و pcb جدا دارند و کلا به عنوان یک هویت مستقل توسط سیستم عامل شناسایی میشن ولی یک برنامه مالتی ترد اینطوری نیست ینی یک پروسس است که تعداد های مختلفی داره : اونقدر تردها استقلال ندارند که پروسس ها استقلال دارند
یکی از برنامه هایی که از Multiprocess استفاده میکنه مرورگر کروم است ینی هر کدوم از این تب هایی که توی کروم داریم یک پروسس جدید است همچنین یک پروسس اصلی داریم که بهش browser میگیم ولی به این تب های پروسس های مختلف renderer میگیم و کارش اینه که html رو بگیره و به اون صورتی که با html تفسیر میشه به ما نشون میده
ممکنه plug-in های مختلفی روی کروم اضافه کرده باشیم که هر کدوم از این plug-in ها هم خودشون به عنوان یک پروسس اضافه میشن به سیستم
مزیتش : sandbox است
توی اپلیکیشن نویسی وقتی sandbox میگیم ینی یه چیزی رو اونقدر خوب جدا کرده باشیم که مشکلاتی که توی sandbox وجود داره به خارج نشت پیدا نکنه و اون برناهم اصلی مشکل پیدا نکنه مثلا اگر یکی از تب ها بلاک شد کل کروممون بسته نمیشه چون کروممون خودش یک پروسس دیگه ای است ولی اگر مالتی ترد بود ممکن بود که اگر یک ترد به یک مشکلی برمیخوره باعث بشه کل پروسسمون هم بسته بشه

Interprocess Communication

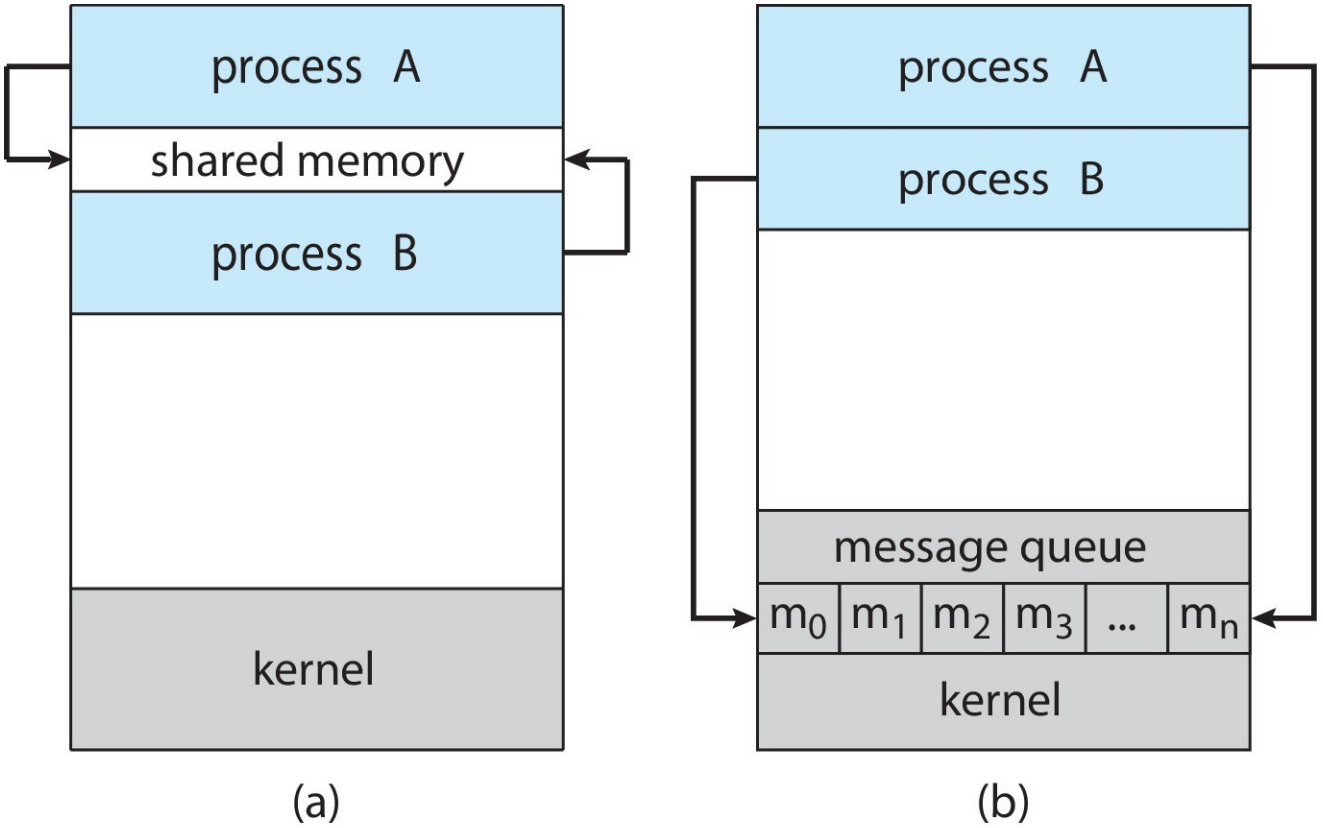
- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing (for small amounts of data)**

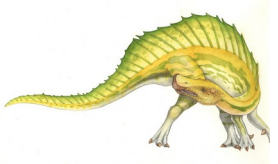


-
وقتی که برنامه Multiprocess داریم خیلی وقت ها نیاز میشه بین این پروسس ها یک ارتباطی رو برقرار کنیم یا یک سری اطلاعاتی بینشون رد و بدل کنیم یا اصطلاحا بینشون شیر کنیم چرا؟
- چون ما Information sharing کلا داریم یعنی یکسری اطلاعاتی داریم که همه پروسس ها باید همزمان از همون اطلاعات استفاده بکنند
- یا اینکه مثلا داریم یک تسکی رو توسط پروسس های مختلف اجرا میکنیم به صورت موازی و علت اینکه Multiprocess کردیم اینه که بالا بردن سرعت محاسبات بوده
- گاهی به قصد modularity ما اصلا Multiprocess کردیم و اینجا نیازه که برای اینکه اون کار اصلی کل اون پروسس انجام بشه بتونیم اطلاعاتی رو بین اینها رد و بدل کنیم
- برای ساده سازی و برای اینکه راحت تحت کنترلمون باشه یه جاهایی نیاز داریم که ارتباطی بین پروسس ها داشته باشیم یا به اصطلاح میگیم یک IPC می خوایم داشته باشیم
دو تا روش IPC داریم که توی این درس مورد بررسی قرار میدیم:
shared memory
message passing

Communications Models

(a) Shared memory. (b) Message passing.



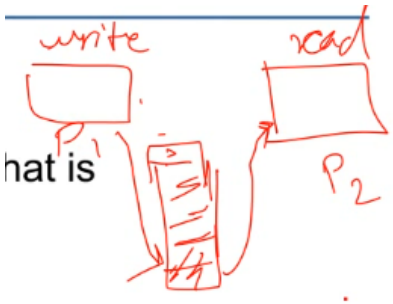


-
مقایسه shared memeory , message passing:
اینجا یک قسمتی از حافظه ادرس فیزیکی یا RAM رو داره نشون میده
دو تا پروسس توی رم لود شده
توی شکل a می بینیم پروسس A , B از طریق shared memory با هم دیگه در ارتباط هستند
ینی یک قسمت از فضای حافظه وجود داره که هر دوشون همزمان بهش دسترسی دارن
اما تو روش Message passing اتفاقی که می افته اینه که ما یکسری پیام می بینیم مثل m0 تا mn و این پیام ها روی کرنل قرار گرفته چرا ؟ ما تو حالت Message passing یک امکاناتی داریم که سیستم عامل برامون فراهم کرده و اجازه می ده یکسری اطلاعاتی از پروسس A به پروسس B متنقل بشه یا برعکس و از طریق یک صفی از پیام ها که خود سیستم عامل داره اینو مدیریت میکنه پس اینجا سیستم عامل وارد عمل شده (ینی مدیریت این پیام ها که داره بین A, B رد و بدل میشه سیستم عامل داره انجام میده) ولی توی روش Shared memory سیستم عاملی نمی بینیم

Producer-Consumer Problem

- Paradigm for cooperating processes:
  - *producer* process produces information that is consumed by a *consumer* process
- Two variations:
  - **unbounded-buffer** places no practical limit on the size of the buffer:
    - Producer never waits
    - Consumer waits if there is no buffer to consume
  - **bounded-buffer** assumes that there is a fixed buffer size
    - Producer must wait if all buffers are full
    - Consumer waits if there is no buffer to consume

-
برای نشون دادن این ارتباط بین پروسس ها یک مسئله معروفی که وجود داره مسئله Producer-Consumer است
مسئله Producer-Consumer ینی یکی از پروسس ها داره یکسری از اطلاعاتی رو تولید میکنه و اون یکی پروسس داره اون اطلاعات رو مصرفش میکنه که خیلی جاها این مسئله یک مسئله پایه است و خیلی جاها ارتباط بین پروسس ها به این صورت است
حالا ممکنه این اطلاعات که توسط یک پروسس تولید کننده داره تولید میشه توی یک بافر نامحدود ریخته بشه و مصرف کننده برشون داره و یا ممکنه bounded باشه ینی همزمان که این p1 , p2 که دارند با هم کار میکنند و همزمان که این p1 داره یکسری اطلاعاتی تولید میکنه و p2 نتونه اونارو همزمان بخونه و این باعث میشه اینجا یک بافری تشکیل بشه ینی p1 اینارو بریزه توی یک بافر و p2 اینارو از توی بافر برداره و بخونه حالا اگر سرعت قرار گرفتن اطلاعات توی بافر بیشتراز سرعت خوندن p2 از بافر باشه و ما اگر بافر bounded داشته باشیم ممکنه به مشکل بخوریم
ولی توی unbounded تولید کننده باید بتونه مدام بدون صبر کردن توی یک بافری بریزه و مصرف کننده هر وقت دلش خواست ازش برداره ولی اگر p1 چیزی نریخته بود توی بافر p2 باید صبر بکنه
نکته: توی حالت bounded buffer هر کدومشون ممکنه صبر بکنه ولی توی حالت unbounded buffer فقط مصرف کننده است که گاهی مجبور صبر بکنه که بافر پر بشه



Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- **Benefits:**
  - Faster than message passing
- **Disadvantages:**
  - Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.

-
معایب:
اگر از shared memory استفاده بکنیم سختی کار با خودمونه ینی بحث هایی مربوط به synchronize رو خودمون باید مدیریت کنیم ولی توی massege passing می تونیم خیلی راحت استفاده بکنیم و اصلا کاری به بحث synchronize نداریم چون سیستم عامل داره اینو برامون مدیریت میکنه
مزایا:
چون shared memory خیلی ساده تره ممکنه برای استفاده های خیلی ساده ای سریعتر باشه چون اون لایه های برنامه نویسی که سیستم عامل این وسط اضافه کرده دیگه توی این نداریم پس برای سرعت بهتره از shared memory استفاده بکنیم و برای راحتی کار بهتره از massege passing بریم

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solution is correct, but can only use `BUFFER_SIZE-1` elements

-
مثالی از bounded buffer :
سایز بافر 10 است
چون از shared memory داریم استفاده می کنیم کنترل بافر با خودمون است پس از دوتا ایندکس استفاده می کنیم ینی :

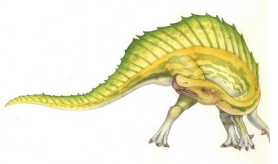
int in =0

int out=0

که بتونیم روی ورودی و خروجی بافر کنترل داشته باشیم :

Producer Process – Shared Memory

```
item next_produced;

while (true) {
   /* produce an item in next produced */
   while (((in + 1) % BUFFER_SIZE) == out)
       ; /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
}
```



-
برنامه ای برای مدیریت shared memory
برنامه پروسس تولید کننده:
باید همیشه حواسش به این باشه که بافر پر نشده باشه چون بافر bounded است واسه همین in+1 کرده که توی while نوشته شده که شرط توی while مال زمانی که پره و اینقدر صبر میکنه تا در نهایت یکی از خونه های بافر توسط مصرف کننده خونده بشه و خالی بشه تا دیگه شرط توی حلقه برقرار نباشه
در اینصورت پروسس تولید کننده میتونه مقدار جدید رو توی بافر بریزه :
buffer[in]=next_produced

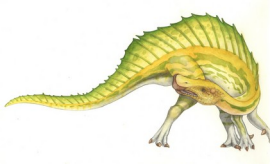
Consumer Process – Shared Memory

```
item next_consumed;

while (true) {
    while (in == out)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next consumed */
}
```

-
کد مصرف کننده برعکسه
وقتی که in با out برابر بشه ینی چیزی توی بافر نیست پس اونقدر باید صبر بکنه که توی بافر پر بشه

What about Filling all the Buffers?

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffers.
- We can do so by having an integer **`counter`** that keeps track of the number of full buffers.
- Initially, **`counter`** is set to 0.
- The integer **`counter`** is incremented by the producer after it produces a new buffer.
- The integer **`counter`** is and is decremented by the consumer after it consumes a buffer.



-
به جای اینکه با in , out بافر رو چک کنیم می تونیم با یک counter هم بافر رو چک کنیم

Producer

```
while (true) {
    /* produce an item in next produced */

    while (counter == BUFFER_SIZE)
        ; /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Consumer

```
while (true) {
    while (counter == 0)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
        counter--;
    /* consume the item in next consumed */
}
```

Race Condition

- **`counter++`** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **`counter--`** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 - 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}

Race Condition (Cont.)

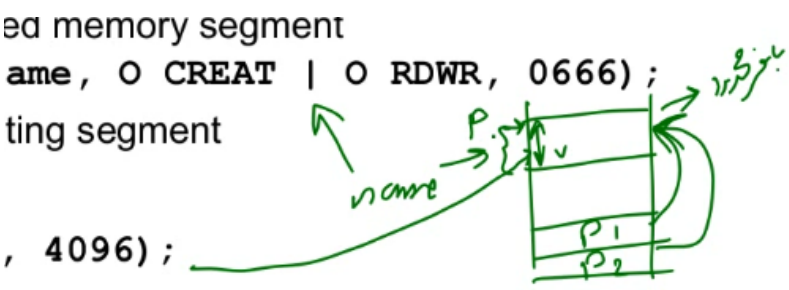
- Question – why was there no race condition in the first solution (where at most N – 1) buffers can be filled?
- More in Chapter 6.

Examples of IPC Systems - POSIX

- POSIX Shared Memory
  - Process first creates shared memory segment
    `shm_fd = shm_open(name, O CREAT | O RDWR, 0666);`
  - Also used to open an existing segment
  - Set the size of the object

    `ftruncate(shm_fd, 4096);`
  - Use `mmap()` to memory-map a file pointer to the shared memory object
  - Reading and writing to shared memory is done by using the pointer returned by `mmap()`.

-
یک مثالی از POSIX Shared Memory می بینیم اینجا:
اول یک دیتا استراکچری برای Shared Memory در نظر گرفته میشه که باید با یک نام Shared Memory رو باز کنیم و همین نام استفاده میشه برای هر پروسسی که بخواد از این Shared Memory استفاده بکنه
پس Shared Memory یک نام داره و پروسس هایی که می خواین با هم دیگه Shared Memory داشته باشن باید از همین نام مشترک استفاده بکنن
بعد از اون باید مشخص بکنیم که این فضای حافظه که برای Shared Memory داریم اندازش چقدر باشه ینی چقدر از حافظه رم رو می خوایم اختصاص بدیم به Shared Memory
به یک قسمتی از حافظه Shared Memory می گیم و دوتا پروسس به اون قسمت از حافظه دسترسی دارند و اینکه این تیکه از حافظه چقدر باشه رو توسط دستور ftruncate مشخص کردیم
الان این تیکه رو که Shared Memory در نظر گرفتیم شده یک بافر محدود و ما برای مدیریتش نیاز داریم که یک ایندکسی ازش نگه داریم مثلا یک پوینتر p براش انتخاب میشه که ادرس اول این فضای ادرس است و بعد ما باید خودمون این پوینتر رو باید ببریم جلو پس این پوینتر رو نیاز داریم که تابع mmap این پوینتر قسمت اول حافظه رو بهمون میده و بعد از اون با استفاده از پوینتر دسترسی مستقیم به این قسمت حافظه داریم
شکل این قسمت:



IPC POSIX Producer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```

اینجا مربوط به مدیریت ایندکس فضای ادرس مشترک است

-
یک کد ساده از Shared Memory :
راحت با ptr که پوینتر ما هست اینجا می تونیم کار بکنیم

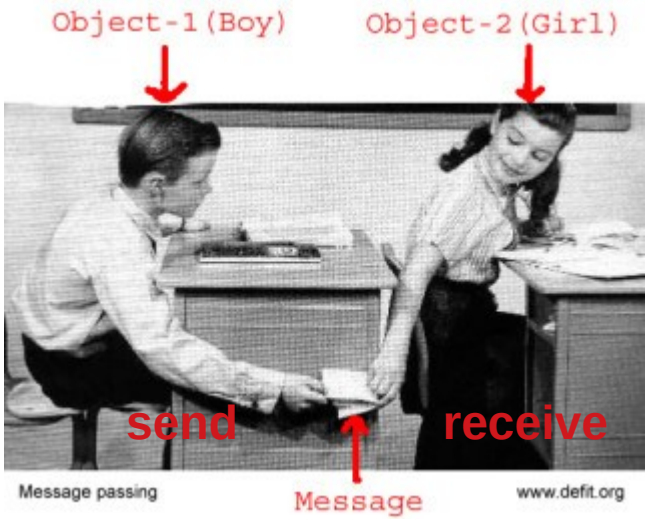
IPC POSIX Consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* open the shared memory object */
   shm_fd = shm_open(name, O_RDONLY, 0666);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

   /* read from the shared memory object */
   printf("%s",(char *)ptr);

   /* remove the shared memory object */
   shm_unlink(name);

   return 0;
}
```

مصرف کننده IPC POSIX

IPC – Message Passing

- Message system – processes communicate with each other without sharing the same address space directly
- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
- The *message* size is either fixed or variable

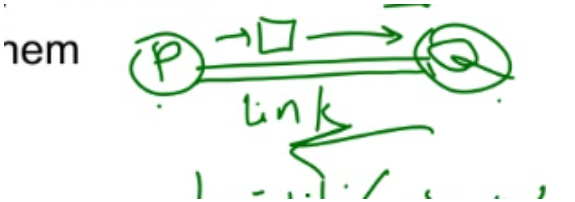


-
: Message Passing
یک ارتباطی بین پروسس ها است و از طریق ارسال پیام می تونن این کارو انجام بدن بدون اینکه یک فضای ادرسی رو با هم دیگه شیر کرده باشند و این هم بخاطر کتابخونه های سیستمه که این امکانات رو به ما میده پس ما پیام رو ارسال میکنیم و یک نفر دریافت میکنه پس برای پیام send , receive داریم و پیامی که ارسال میشه می تونه یک سایز مشخص و فیکسی داشته باشه یا نه و می تونه متناسب با پیام سایز پیام عوض بشه

Message Passing

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

-
نحوه فراهم Message Passing : مشخصاتش:
مهمترین مسئله اینه که وقتی که دوتا پروسس دارند با هم پیام رد و بدل می کنند باید بینشون یک communication link باشه که این پیام ها روی این لینک بتونند ارسال بشند : شکلش بالا:
مسائل و سوالاتی که درباره ی سیاست این کانال ارتباطی داریم:
- اصلا چجوری لینک رو بسازیم؟
- ایا یک لینک می تونه برای بیشتر از دوتا پروسس باشه؟
- اگر p به q فرستاد ایا q هم می تونه به p بفرسته روی همین لینک یا باید یک لینک جدا بسازه روش؟
- ظرفیت این لینک چقدره؟
- ....
جواب اینا وابسته به اون کتابخونه ای که ما برای Message Passing داریم استفاده میکنیم و وقتی که داریم کد می نویسیم برای اون کتابخونه باید دقت کنیم که چجوری این پارامترها رو تنظیم کنیم که اون جوری که میخوایم ازش استفاده بکنیم



Message Passing model

- **Physical link**
  - Shared memory, Hardware bus, Network
- **Logical link**
  - Naming
    - Direct (name the process), indirect (name the mailbox or link)
  - Synchronization
    - Block, Non-block
  - Buffering
    - Zero, bounded, un-bounded

چندتا مشخصه برای مدل Message Passing تعریف میکنیم:
لینک فیزیکی : ینی اون چیزی که موجوده ینی وقتی ما داریم پیام رو ارسال میکنیم یه جایی باید این پیام بره تا اون طرف دریافت بشه مثلا اگر ما بین دوتا پروسس روی یک سیستم داریم این کارو انجام میدیم ممکنه از Shared memory استفاده بشه - و اگر داریم بین دوتا پروسس روی شبکه این هارو رد و بدل میکنیم ینی دوتا پروسس روی دوتا شبکه مختلف هستند پس می تونیم از network استفاده بکنیم یا ممکنه توی بعضی از سیستم ها از باس سخت افزاری برای پیام استفاده بشه
لینک منطقی : ینی اون چیزی که منطقا برامون اون سیستم یا کتابخونه می سازه که از طریقش بتونیم کار بکنیم اون به چه صورت است و چیا رو در نظر گرفته:
1- روش naming : مثلا p میخواد برای q پیام بفرسته ایا باید توی کدی که می نویسه ادرس q یا اسم q رو مشخص بکنه تا پیام براش ارسال بشه به این حالت میگیم دایرکت ینی نامگذاری پروسس ینی وقتی که ما می خوایم پیام رو بفرستیم تابعی که استفاده می کنیم مثلا f(msg,q) ینی این f رو پروسس p داره می فرسته که واین تابع محتوای پیام هست و q نام اون پروسسی که میخواد براش پیام رو ارسال بکنه ، یه موقعی هم هست کانالی که بین p,q هست نامگذاری میکنیم که به این حالت میگن indirect توی این حالت باید بگه f(msg , name canal) ینی اسم کانال رو بهش میده توی بخش دومش (پس توی این حالت قبل از اینکه پیام رو ارسال کنیم باید این کانال رو بین p,q بسازیم)
2- روش synchronization : مثلا اگر p یک پیامی رو فرستاد برای q ایا باید صبر بکنه که q پیام رو دریافت بکنه و بعد پیام های بعدی رو بفرسته یا نه می تونه رد بشه و همین جوری پیام هارو بفرسته و q هر وقت دلش خواست پیام هارو برداره - برعکسش هم هست که ما بیشتر اینو دیدیم پس توی q اگر تابع رید رو فعال میکنیم یا recevie رو فعال میکنیم اگر این اونقدر منتظر موند که بالاخره p براش پیام رو بفرسته و پیام رو گرفته و رد شد به این حالت میگیم block شده ولی اگر نه ینی رسید به این خط و دید پیامی نیست و ازش رد شده این حالت non-block میشه
3- روش buffering : اطلاعاتی که داره بین p,q رد و بدل میشه ایا اول از p می ره توی یک بافری و بعد توسط q دریافت میشه اگر بافر نداشته باشیم کلا بهش می گیم zero buffer ینی این p به محض اینه یک اطلاعاتی رو میده این q باید برش داره ولی اگر یک بافر محدود داشته باشیم بهش میگیم bounded و اگر بافر نامحدود داشتیم میگیم un-bounded - با امکاناتی که ما داریم تقریبا همیشه روشمون bounded است

Producer-Consumer: Message Passing

- Producer

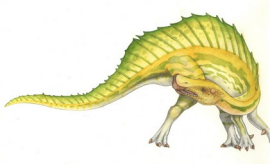
```
message next_produced;
while (true) {
 /* produce an item in next_produced */

 send(next_produced);
}
```

- Consumer

```
message next_consumed;
while (true) {
 receive(next_consumed)

 /* consume the item in next_consumed */
}
```

Pipes

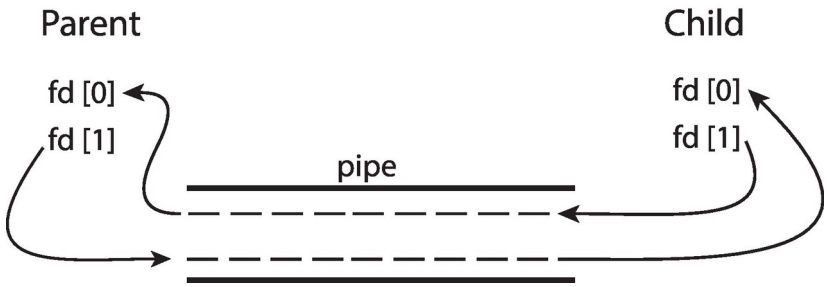
- Acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
  - Can the pipes be used over a network?
- **Ordinary pipes** – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- **Named pipes** – can be accessed without a parent-child relationship.



-
چندتا از روش های Message Passing:
pipes:یک pipe بین دوتا پروسس ایجاد بشه ینی یک لوله ای بین دوتا پروسس ایجاد بشه و اطلاعات میاد توی این لوله و توسط اون یکی پروسس برداشته می شه
همان سوالاتی که راجع به Message Passing داشتیم راجع به pipe ها هم داریم
پایپ لینوکس half duplex است
به طور کلی دو مدل پایپ داریم:
Ordinary pipes : این ها فقط توسط والد و فرزند قابل دسترسی هستند ینی ارتباط بین دوتا پروسس باید والد و فرزندی باشه و نمی تونه هر نوع پروسسی باشه
ولی توی Named pipes می تونه بین هر دو پروسسی اطلاعات رد و بدل بشه

Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes



- Windows calls these **anonymous pipes**

-
این روش مثل مسئله producer-consumer کار می کنه
ینی یک طرف مثل والد می تونه اطلاعات رو بذاره و فرزند برداره یا برعکس ینی فرزند بذاره و والد برداره
اینجا برای پایپ دوتا انتها تعریف میکنیم : انتهایی که برای خواندن است و انتهای که برای نوشتن است ینی دوتا پوییتر داریم :
fd[0] و fd[1] و از fd[1] همیشه برای قرار دادن توی پایپ استفاده می کنیم ینی رایت کردن و fd[0] برای خوندن

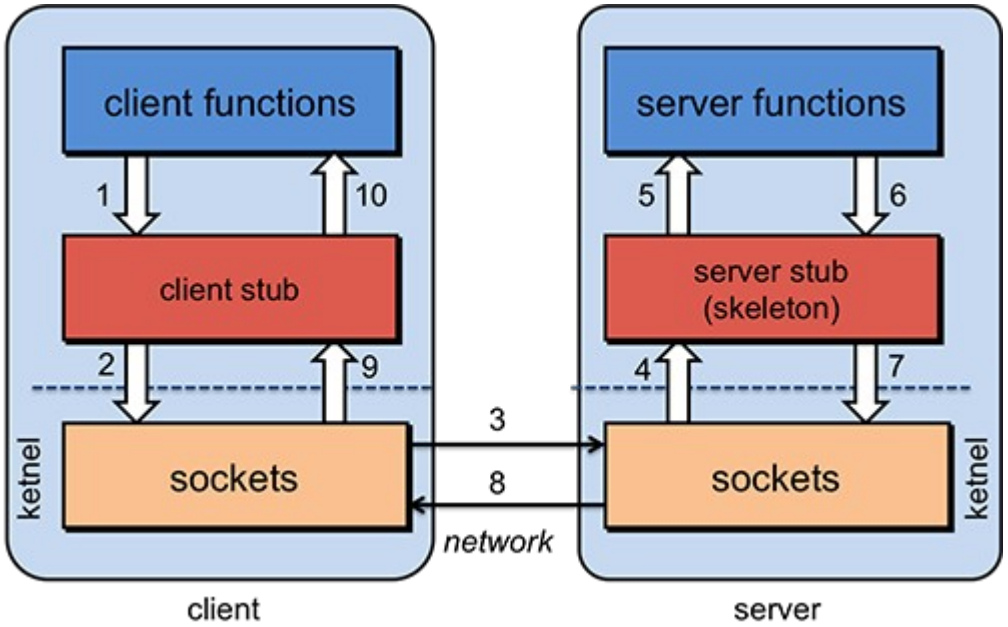
Named Pipes

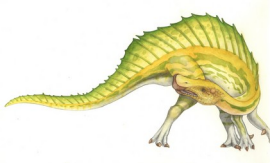
- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems

-
Named Pipes ها امکانات بیشتری دارند بخاطر اینکه ما می تونیم بین هر دو پروسسی ایجادشون کنیم و نیاز نیست حتما ارتباط والد و فرزندی باشه
و ارتباط هم دو طرفه است ینی هر دو طرف می تونن پیام ارسال کنند
و تعداد زیادی پروسس می تونن همزمان از طریق پایپ با هم در ارتباط باشند ینی Named Pipes می تونه بین بیشتر از دوتا پروسس هم ارتباط برقرار بکنه

Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls (RPC)





--

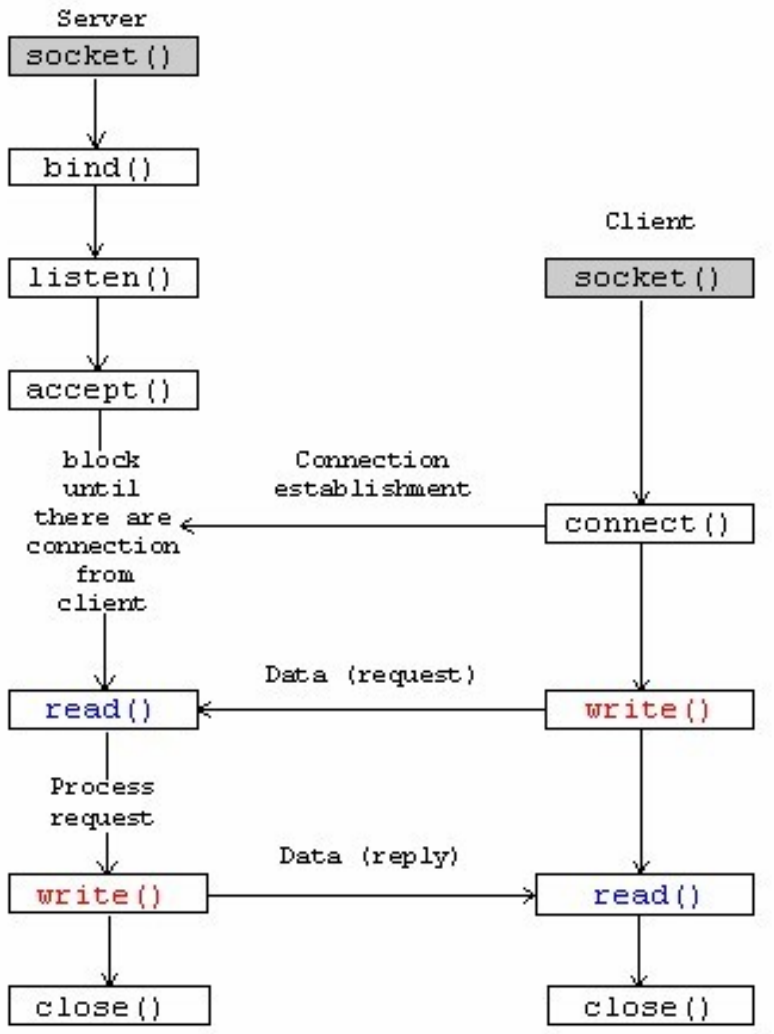
Sockets

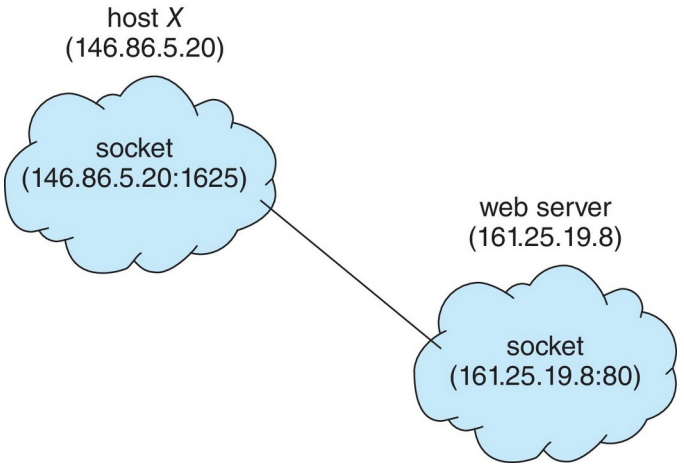
- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- All ports below 1024 are ***well known***, used for standard services
- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running

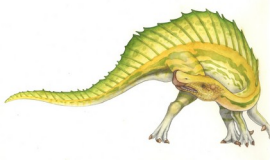
- Three types of sockets
  - **Connection-oriented** (**TCP**)
  - **Connectionless** (**UDP**)
  - **`MulticastSocket`** class– data can be sent to multiple recipients
- Consider this “Date” server in Java:

--

Socket Communication



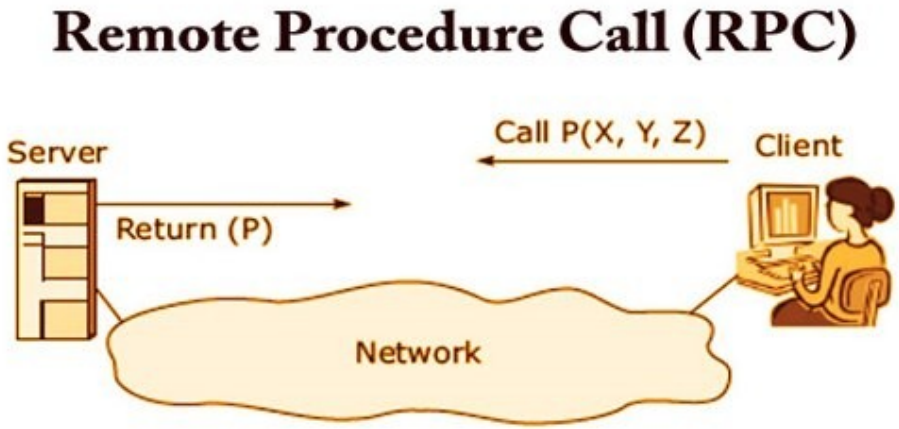


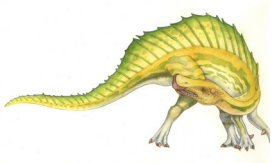


سوکت ارتباط

Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - Again uses ports for service differentiation
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and **marshalls** the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- On Windows, stub code compile from specification written in **Microsoft Interface Definition Language** (**MIDL**)



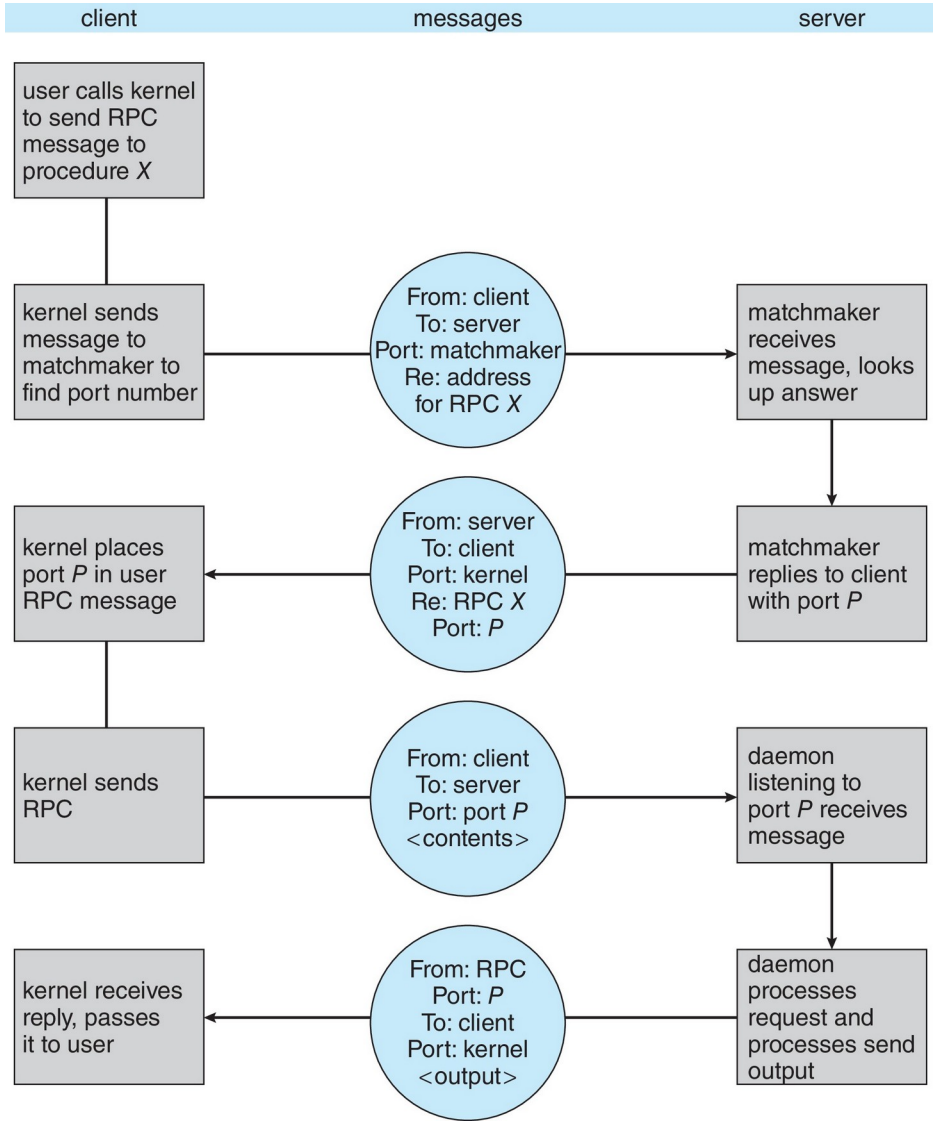


-
Remote Procedure Calls ینی یک فانکشنی از یک سیستم دیگر فراخوانی بشه
مثلا توی عکس پایین:
یک کلاینت داریم و یک سرور و روی سطح شبکه هم از هم جدا هستند و این کلاینت می تونه یکی از فانکشن های سرور ینی مثلا فانکشن p را فراخوانی بکنه ینی کد این p توی سرور است و این کلاینت درخواست می کنه که این p را براش فراخوانی کنه و این ورودی ها ینی x,y,z رو خود کلاینت بهش میده و سرور کاری که میکنه اینه که این p رو که توی ماشین خودش قرار داره با اون ارگومان هایی که کلاینت بهش داده توی خودش اجرا میکنه و بعد خروجی رو برمی گردونه به کلاینت پس مثل اینه که کلاینت خودش این فانکشن رو داشته باشه ولی در عمل اینطوری نیست و این فانکشن توی سرور است و سرور داره اینو براش اجرا میکنه
در همچین نوع ارتباطی ما نیاز داریم که کتابخونه های لازم برای ایجاد ارتباط RPC بین کلاینت و سرور نصب شده باشه
کار Stubs : ترجمه ی یکسری ارتباطات بین کلاینت و سرور است
کار دیگه که Stubs انجام میده marshalls کردن اطلاعات است ینی الان داریم یه سری پارامتر رو بین دوتا ماشین که متفاوت هستند رد و بدل میکنیم ؟؟؟

Remote Procedure Calls (Cont.)

- Data representation handled via **External Data Representation** (**XDL**) format to account for different architectures
  - **Big-endian** and **little-endian**
- Remote communication has more failure scenarios than local
  - Messages can be delivered ***exactly once*** rather than ***at most once***
- OS typically provides a rendezvous (or **matchmaker**) service to connect client and server

Execution of RPC



-
مثال
از بالا شروع میشه
کلاینت یک پیام rpc رو می فرسته که میخواد چه فانکشنی رو فراخوانی بکنه
بعد اینجا از ماژول matchmaker استفاده میشه که بین کلاینت و سرور است که یکسری از تفسیرهارو این ماژول انجام می ده
.....

Examples of IPC Applications

- X Server and client
  - Unix-domain sockets, named pipes, …
- Piping commands in shell
  - pipe
- Database server
  - Socket, RPC
- Web services
  - Kind of RPC
- …

-
مثال هایی از IPC:

End of Chapter 3