**Program**

* Je subor obsahujuci mnozinu usporiadanych krokov, instrukcii veducich k dosiahnutiu zelaneho ciela, spravania

**Proces**

* Je vykonavany program s pridelenymi prostriedkami vypoctoveho systemu
* Je vykonavany program s:
  + Vyjrademy, adresnym priestorom
  + Stavom (register, pc, zasobnik)
  + Pridelenymi I/O zariadeniami
  + Otvorenymi subormi

**Život bez procesov**

* Vykonavanie jednotlivych uloh systemu by prebiehal ako nekonecna slucka. Rad za radom by sa vykonavali podprogramy
* Programator by musel sam vzdy brat do uvahy ostatne aplikacie. Musel by sa starat o dobrovolne prerusovanie cinnosti
* Prakticky nerealne riesit pristupove opravnenia a bezpecnost vypoctoveho system
* Nemozne bezpecne riesit riadenie prvilegovaneho a neprivilegovaneho rezimu
* Pre cloveka nieje jednoduche riesit problem paralelne. Navrh aplikacie ako pozname by nebol mozny
* Procesy musia byt skompilovanou sucastou OS.

**Zivot s procesmi**

* Procesny model definuje process ako sekvencnu pracu a teda umoznuje cloveku z komplexneho problem spravit mnozinu jednoduchsich sekvencnych problemov
* Jasne rozdelenie privilegovaneho a neprivilegovaneho rezimu prace
* V tomto pohlade process vnima CPU akoby bol cely jeho. Preto sa programator moze venovat navrhu svojej aplikacie
* OS zarucuje ze procesy sa postupne striedaju na CPU v chrranenom prostredi
* Process je model resp. abstrakciou vo vypoctovom systeme

**Procesny model**

* Abstrakcia od riadenia pristupu ku zdrojom vypoctoveho system
* Process je izolovany na vypoctovom systeme (neprivilegovany rezim)
* Process a jeho prostredie je chraneny OS
* Process sa nemusi delit o prostriedky
* Striedanie procesov na cpu riadi os
* Programator sa nemusi zaoberat preruseniami
* Pridelovanie pamate a prostriedkov riadi OS
* Umoznuje multiprogramming

**Paralelizmus**

* V jednom okamihu je vykonavany na jednom cpu prave jeden process
* Cas vykonania jedneho procesu je niekolko desiatok milisekund
* V case sa procesy na cpu striedaju
* **Pseudo-paralelizmus**
  + Caste striedanie procesov na jednom cpu budi dojem, ze kazdy process sa vykonava sucasne. V takom pripade hovorime o pseudo-pararelnom spracovani

**Multiprogramming**

* Je udrziavanie viacerych procesov … idk

CPU utilizacia = 1 – pˆn

**Reprezentacia procesu v OS**

* Process table
  + Je tabulka OS, kde su ulozene vsetky vykonavane procesy
* Process control block (PCB)
  + Je jeden zaznam v process table, v ktorom OS udrzuje vsetky potrebne informacie o procese
* Process context
  + Je aktualny stav procesu, ktory sa odklada do PCB v pripade preplanovanie procesu
* Process tree
  + V unix OS procesy vznikaju inymi procesmi. Preto vznika vztah rodic dietam kde rodic si pamata identificatory svojich deti. Takymto sposobom su procesy udrzovane aj v strome procesov. Windows nema takyto concept

**Vznik procesu**

* Pri spusteni OS
  + Bezne pouzivana metoda vo vnorenych systemoch
  + Pri starte OS vzdy vznikne aspon jeden process
  + Unix tento process vola init process
* Systemovym volanim
  + Prostrednictvom ineho procesu
  + Poziadavkou pouzivatela (napriklad cez shel – ls -la)
* Batch Jobom (napriklad planovanou uzavierkou prevadzky sa vytvori Batch uloha na spracovanie trzby)
* **Systemove volanie**
  + Fork() a exec() – Unix
    - V unixe process vznika systemovym volanim fork(). Fork() vytvori presnu kopiu procesu so vsetkymi pridelenymi prostriedkami. Pamat program nieje nutne kopirovat a rodic s dietatom ju zdielaju. Ak chceme spustit iny program Unix pouziva volanie exec(). Exec() nahradi aktualny program procesu novym programom
  + Spawn() – windows
    - Windows pouziva volanie CreateProcess()- Toto volanie na zaklade parametrov vytvori uplne samostatny process
  + Copy-on-write
    - Napriek tomu, ze pocas volania fork() nedochadza ku kopirovaniu program stale je nutne kopirovat data procesu. Tato operacia je tiez nakladna a preto v unix.e sa pouziva metoda copy-on-write. V principe sa pamat dat nekopiruje kym do nej detsky process nechce zapisat
* **Zanik procesu**
  + Dovrovolny zanik
    - Standardne ukoncenie program
    - Standardne ukoncenie s chybou
  + Nedobrovolny zanik
    - Ukoncenie s fatalnou chybou (Segmentation fault, division by zero)
    - Ukoncenie inym procesom (Kill)
* Zanik procesu – systemove volanie
  + Exit()
    - Pre dobrovolne ukoncenie procesu sa pouziva systemove volanie exit()
* Vykonavanie procesu
  + Process po vytvoreni prechadza do fazy vykonavania program. Pocas vykonavanie moze dosiahnut nasledovne stavy
    - Running – process je vykonavany na cpu
    - Ready – process je pripraveny na vykonavanie. Process je planovacim algoritmom vlozeny do poradovnika na vykonavanie
    - Blocked – process je blokovany cakanim na udalost. Process nie je v poradi na vykonavanie

1. Process je blokovany lebo caka na udalost
2. Process je neplanovany OS
3. Process je naplanovany OS
4. Process je zobudeny po prichode udalosti

**THREADS (Vlákna)**

* Process
  + Je jedno alebo viacej vykonavanych thread-ov program s pridelenymi prostriedkami vypoctoveho system
* Thread
  + Je vykonavany program s vlastnym stavom (register, PC, zasobnik)
  + Vlakno je mozne vidiet ako process vykonavany v kontexte procesu
  + Process je mozne v tomto koncepte udrziavat viacero vlakien
    - V jednom momente len jedno vlakno je vykonavane na cpu
    - Ostatne vlakna mozu byt pripravene alebo blokovane
  + Vlakna su v podstate velmi lahke procesy ktore zdielaju v ramci jedneho procesu spolocny adresny priestor, subory, signaly…
  + Vyhody pouzitia vlaken:
    - Rozdelenie program do este mensich sekvencnych uloh
    - Vlakna navzajom mozu komunikovat a synchronizovat sa prostrednictvom spolocnej pamati. U procesov to je tiez mozne ale omnoho nakladnejsie
    - Preplanovanie vlakna inym vlaknom v ramci procesu je lacnejsie nakolko nieje nutne pri zmene kontextu vymenit pamat program
    - Cpu s multithreading umoznuje jednoduchsi pristup k vlaknu
  + **Komplikacie**
    - Spravanie fork() – ked vytvaram novy process vytvorim ho aj so vsetkymi existujucimi vlaknami?
    - Zdielanie prostriedkov – ak viacej vlakien pracuje so suborom. Co sa deje ak jedno vlakno subor zavrie?
    - Alokacia pamate – co ak viacero vlakien sucasne poziada system o pridelenie pamate
    - Praca so stackom – ak vlakno potrebuje vacsi zasobnik ako vie OS urcit kto ho potrebuje
    - Globalne premenne – nemaju chraneny pristup
    - Signalizacia – kto reaguje na prichod signal od OS ? vsetci alebo aktualne vlakno?
    - Kniznice – kniznice musia byt implementovane tak aby podporovali threading.
* **Klasicky thread model**
  + Process je kontajner udrziavaci prostriedky
  + Thread ma program, register, pc, stack
  + **Thread control block**
    - Je struktura, v ktorej OS udrzuje vsetky potrebne informacie o vlakne
* **POSIX thread model**
  + POSIX vlakna standardizuju a rozsiruju klasicky model vlaken
  + POSIX definuje cez 60 volani suvisiacich s vlaknami
* **Reprezentacia vlakien v OS**
  + User space threads
  + Kernel space threads
* **User space threads**
  + Kniznica s funkciami, ktore neobsahuju systemove volanie
  + Kernel OS nie je volany – rychle operacie nad vlaknami
  + Tabulka vlakien je udrziavana v kontexte procesu
  + Kernel OS nemusi podporovat vlakna
  + Kazdy process moze implementovat vlastny planovaci algoritmus
  + Neexistuje moznost prepinat vlakna pomocou OS
* **Kernel space threads**
  + Kernel implementuje podporu vlaken
  + Operacie nad vlaknami obsahujju systemove volanie
  + Kernel sprostredkuje planovanie vlaken
  + Kernel umoznuje prepnutie vlakna bez potreby volania yield()

**PLANOVANIE PROCESOV**

**Scheduler (Planovac)**

* Je cast OS, ktoreho ulohou je usporiadanie pripravenych procesov, tak aby splnil definovane kriteria planovania na vybranom idk…

**Kedy OS potrebuje planovat?**

* Vykonavany process skoncil
* Vykonavany process vytvoril novy process
* Vykonavany process je blokovany systemovym volanim
* HW prerusil CPU
  + I/O zdrojom prerusenia
  + Periodicky casovac zdrojom prerusenia
* To, ze doslo k udalosti kedy je mozne planovat nemusi vyustit k prepnutiu aktualne vykonavaneho procesu

**Klasifikacia planovacich algoritmov podla reakcie na HW prerusenie**

* **Preemptive** – planovanie umoznuje prerusenie vykonavanej ulohy planovacom
* **Non-preemptive –** planovanie dovoluje nahradit vykonavany process len ak sa on dobrovolne rozhodne alebo ak je blokovany systemovym volanim
* **Batch** – batch OS bezne nema aktivnych pouzivatelov a preto nieje potrebna preempcia. Ak je pouzite preemptivne planovanie je nastavene na dlhe casove useky. Dolezitymi kriteriami planovania su throughput

**Klasifikacie process podla vazby**

* CPU-bound – vypoctovo intenzivne
* I/O bound – I/O intenzivne

**Kriteria planovania**

* Spolocne
  + Ferovost – kazdy process by sa mal dostat k CPU
  + Vynucovanie politiky – politika planovania je dodrziavana vzdy
  + Vyvazenost – cely system je rovnako zaneprazdneny
* Batch systemy
  + Throughput – maximalny pocet uloh spracovanych za cas
  + Turnaround time – doba vykonania ulohy je co najkratsia
  + Cpu utilizacoa – cpu je vytazene neustale
* Interaktivne systemy
  + Doba odozvy – poziadavky spracovane co najrychlejsie
  + Proporcnost – splna ockavanie pouzivatela
* Real-Time systemy
  + Dodrzanie doby odozvy – zamedzenie strate dat
  + Predvidatelnost – planovac sa sprava rovnako za rovnakej situacie

**Nepreemptivne algoritmy**

* First come first served (FCFS, FIFO)
  + Jednoduchy, ferovy algoritmus. Prepne na process ktory prisiel ako prvy. Process je vykonavany kym nieje blokovany. Potom je zaradeny na koniec radu. Nevyhodou je ze CPU bound ulohy su zvyhodnene oproti I/O bound
* Shortest Job First (SJF)
  + Potrebuje presny odhad vykonavania ulohy nakolko ta s najkratsim casom je usporiadana na zaciatok radu uloh. Skracuje turnaround time. Optimalny iba ak vsetky ulohy su spustene simultanne (naraz). Pri dostatocnom prude kratkych procesov dlhe prakticky nedostavaju CPU

Preemptivne algoritmy

* Shortest remaining time
  + Preemptivna obdoba SJF. Uloha ktora ma najkratsi zostavajuci cas do ukoncenia ma prednost. Skracuje turnaround time. Najoptimalnejsi je pre systemy, kde vsetky procesy sa spustia pris tarte system. Nove kratke ulohy maju lepsiu sancu dostat sa k cpu, co zvysuje ich odozvu
* Round-robin
  + Jeden z najstarsich najjednoduchsich najferovejsich a najviac pouzivanych algoritmov. Preemptivna podoba FCFS. Dolezitym prvkom je pridelenie casoveho quanta kazdemu z procesov
* Prioritne planovanie
  + Jednotlive procesy maju roznu dolezitost ktora je zohladnena prioritou. Cim vyssia priorita tym je uloha vyssie v rade, nekonecno dlho vykonavana uloha s vysokou prioritou by vyhladovala ostatne ulohy a preto je priorita ulohy s casom postupne znizovana alebo ma uloha pridelene quantum. Priorita moze byt pridelena staticky alebo dynamicky. Prioritu si moze vzajomne kooperujuce ulohy odovzdavat. Je bezne ulohy s rovnakou prioritou planovat pomocou round-robin
* Loteriove planovanie
* Garantovane planovanie
  + Algoritmus ktoreho ulohou je zabezpecit aby vsetky procesy v systeme dostali rovnaky pridel casu vykonavania na cpu. Kazdy process dostane 1/n vykonu CPU. N úloh
* Planovanie s primeranym podielom
  + Obdoba garantovaneho planovania s dorazom na rovnomerne rozlozenie CPU pre pouzivatelov alebo skupiny procesov. Napriklad ak jeden pouzivatel ma 10 procesov a druhy ma 3 tak s garantovanym planovanim by aktivnejsi pouzivatel dostal viac CPU. Aplikovanim primeraneho podielu by obaja pouzivatelia mali dostat rovnaky podiel CPU

**Planovanie vlake**

* User threads
  + Nakolko os nema vedomost o vlaknach planovac ich nedokaze zohladnit. Ak sa vlakna casto striedaju pri cinnosti tak system bla bla