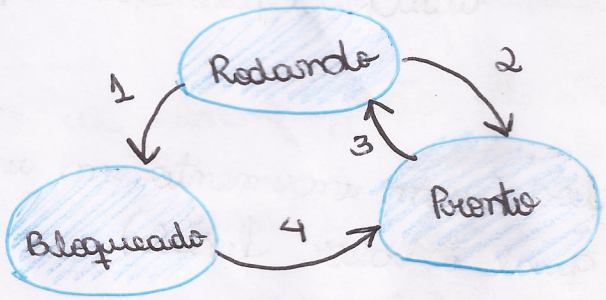
Processos

→ de um programa em execução (*)[1]

→ estados



→ alarme da bateria
outro processo c/ ↑ prioridade

②
1. Entrada de dados (espera)
2. Escalonador escolhe outro (*)[2]
3. Escalonador escolhe ele (*)[3]
4. Entrada de dados (concluída)

(*)[2]: tempo demais usando o processador
(*)[3]: pouco tempo usando o processador

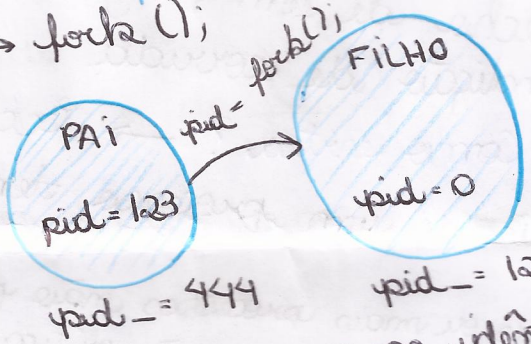
→ processo com muito I/O tem maior prioridade

→ contexto:

variáveis	*p.data
$PC	*p.text
$sp	Estado (PSW)
arquivos	...

→ hierarquia

→ fork();

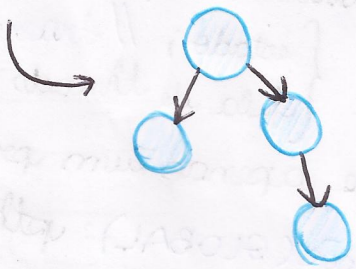


→ programa VS processo:

programa: sequência de instruções
processo: um programa em execução

(*) Cria um processo idêntico ao pai.

ex: fork(); ← Pai
fork(); ← Pai e 1º filho
... ← Pai, 1º f e 2º f



→ função wait()

um filho específico: ponteiro!

wait(pid, int* status)

pid do filho

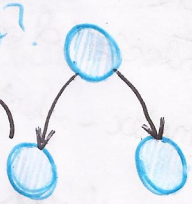
O filho segue as instruções do fork que o criou

WIFEXITED(status) → retorna 0 se tudo ocorreu naturalmente

→ retorna 1 se tudo aconteceu corretamente

→ como pegar o retorno?

status = WEXITSTATUS(status)



ex2:

```
fork();
int pid = fork();
if (pid > 0) {
    fork();
else {
    if (pid == 0) {
        nop
}
```

→ esperando vários filhos:

```
// ... // final do código do pai
while(wait(NULL)>0);
```

→ outra forma:

```
int filhos_pid[n];
for(...){
    filhos_pid[i] = fork();
// ... // final
waitpid(filhos_pid[i], &status, 0)
```

→ Condição de corrida

acesso a dados compartilhados: um incremento em uma variável global (não se sabe qual o valor final).

→ Região Crítica

trecho de um programa que ocasionar uma condição de corrida.

↳ Como evitar? → Exclusão mútua: impedir que mais de um processo tenha acesso

① Dois ou mais processos não podem estar simultaneamente dentro de suas regiões críticas correspondentes

② Nenhum processo que esteja rodando fora da sua região crítica pode bloquear a execução de outro

③ Nenhum processo pode ser drigado a esperar indefinidamente para entrar em sua região crítica.

Mutex

{ estado; // travado ou destravado
fila de threads; ↳ bloqueia a thread que deu lock. é enquene

garante que apenas um processo terá acesso à região crítica

→ declaração (GLOBAL): pthread_mutex_t mutex;
→ inicialização: pthread_mutex_init(&mutex)
→ travamento: pthread_mutex_lock(&mutex)
→ liberando: pthread_mutex_unlock(&mutex)
→ destruindo: pthread_mutex_destroy(&mutex).

(*) trylock: tenta travar, sem bloquear

```
//...
pthread_lock(&mutex);
contador++;
pthread_unlock(&mutex);
//...//
```

→ controle da região crítica

Semáforo

{ contador ∈ ℝ+
  fila de threads

garante a sincronização de tarefas

→ Operações:

⊖ wait() { contador > 0 : contador--;
  contador == 0 : bloqueia e enqueue;

⊕ post() { !fila.empty() : libera a thread do início
  empty() : contador++;

(*) trywait : não entra na fila!

Problema N Produtores e N consumidores

→ mutex: controla o acesso ao índice do buffer.
→ sem_t prod: começa com o tamanho do buffer
↳ indica quantas posições vazias existem no buffer
→ sem_t cons: começa com 0 p/ travar no início

Produtor: wait no prod e post no cons
Consumidor: wait no cons e post no prod.

Threads : atividades concorrentes executadas por um processo

Processo

Espaço de endereçamento		
variáveis globais		
arquivos abertos e alarmes		
$PC	$PC	$PC
$Regs	$Regs	$regs
$sp	$sp	$sp
estado	estado	estado
t1	t2	t3

→ Processo VS Thread

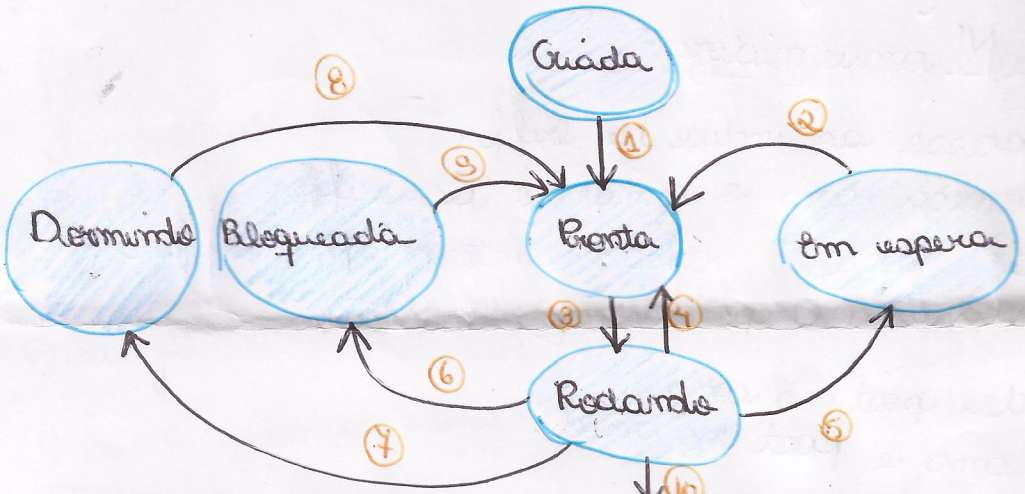
→ criação { cara / barata → recursos { maior / menor

→ troca de contexto { completa / parcial → área da memória { independente / compartilhada

→ comunicação { inter-processo / intra-" → código-fonte { independente / mesmo código

↳ Compartilham variáveis globais (!)

→ Ciclo de vida



1. o programa cria e inicia a thread
2. concluída I/O
3. escalonada
4. Time-out (processador : escalonador)
5. requisição I/O
6. mutex / semáforo
7. sleep()
8. tempo de sleep acabou
9. semáforos ou mutex liberados
10. término da thread

de contexto

duas threads do mesmo processo trocam o core entre si

→ parcial: PC, regs e SP salvos: threads do mesmo processo

↳ total: troca entre threads de processos diferentes.

~ Criando: pthread_create(id, atrib, método, argp) → 0! → cast p/ (void *)

↓

void * método (void * argp)

~ Esperando: pthread_join (id, retorno)

~ Finalizar: pthread_exit (retorno)

↳ *p!

Escalonamento (Linux)

→ threads de um mesmo processo são colocadas na mesma fila

→ podem trocar de fila

→ 1 núcleo : 1 fila

→ prioridade estática: Real-time: 0(↑) a 99(↓)

Normal: 100(↑) a 139(↓)

↳ alterada com nice() → só o root

→ Quantum: tempo máximo de uso do core.

↳ ↑ prioridade: ↑ quantum

- FIFO: ↑ prioridade executa até liberar
- Round-Robin: ↑'s prioridade se alternam na CPU (quantum)
- Deadline: ↑ prioridade: ↓ tempo p/ terminar

Escalonamento (Windows)

→ uso exclusivo do admin

⊛ aplicação com um tempo determinado para terminar

Classe processo (⊖ → ⊕ prioridade):

- Idle
- Below normal
- Normal (padrão)
- Above normal
- High Priority
- RealTime ⊛

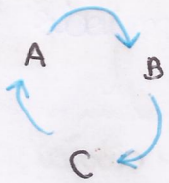
Thread (⊖ → ⊕ prioridade (11)):

- Idle
- Lowest
- Below normal
- Normal (pd)
- Above normal
- Highest
- Time critical

→ prioridade base: classe { 0 (↓) a 31 (↑)
↳ SO

→ prioridade dinâmica:
↳ prioridade base + boost
↳ apenas threads c/ p.b < 16.
↳ garante a igualdade no uso da CPU.
↳ ↑ I/O : ↑ boost

→ Round-Robin: (alternância no uso da CPU)



pA = pB = pC.