Aula 6

Primitivas de Sincronização

Prof. Sérgio Campos / Prof. Marcus Rocha {scampos,mvrocha}@dcc.ufmg.br

Primitivas de Sincronização

- Lock / Unlock
- Sleep / Wakeup
- Semáforo
- Monitores
- Mensagens
- Sincronização no Linux

Lock / Unlock

- Funcionam como as primitivas vistas até agora

```
mutex m;
...
lock(m);
secao_critica();
unlock(m);
...
```

  - espera ocupada

Sleep / Wakeup

- Evitam espera ocupada: liberam o processador quando o processo não pode prosseguir

P0

```
bed b;
...
if (!dado_pronto)
   sleep(b);
/* dado pronto! */
```

P1

```
...
dado_pronto = 1;
wakeup(b);
...
```

Sleep / Wakeup

```
struct bed {
    wait_queue queue;
}
sleep(b, m) {
  currentproc->state = BLOCKED;
  insert_queue(b->queue, currentproc);
  unlock(m);
  schedule();
  lock(m);
}
wakeup(b);
  p = remove_queue(b->queue);
  if (p != NULL) p->state = READY;
};
```

Lock / Unlock

```
struct better_mutex {
    mutex m;  bed b;  int free;
}
better_lock(m) {
  lock(m->mutex);
  if (!m->free) sleep(m->bed, m->mutex);
  m->free = 0;
  unlock(m->mutex);
}
better_unlock(m);
  lock(m->mutex);
  m->free = 1;
  wakeup(m->bed);
  unlock(m->mutex);
};
```

Semáforos

- São variáveis inteiras que são acessadas ***somente*** através de duas operações *atômicas* P (wait) e V (signal)

```
wait(S) {
  while (S <= 0);
  S--;
}

signal(S) {
  S++;
}
```

Exclusão Mútua usando Semáforos

```
int S = 1;

lock(S) {
  wait(S);
}

unlock(S) {
  signal(S);
}
```

Monitores

- As primitivas de sincronização vistas até agora são de baixo nível
  - Podem levar a erros se usadas de maneira errada
- Monitores resolvem este problema fornecendo primitivas de mais alto nível

Monitores

- Monitores são *módulos* de programação
  - Variáveis
  - Funções
- Variáveis só podem ser acessadas por funções do módulo

- Somente uma função pode ser executada a cada instante (em exclusão mútua).

Mensagens

- As primitivas vistas até agora exigem memória compartilhada, que nem sempre está disponível
  - Multiprocessadores modernos;
  - Sistemas distribuídos.
- Mensagens exigem apenas um canal qualquer de comunicação: ***mais gerais***
- Não se assume nada:
  - Tempo de envio;
  - Garantia de chegada;
  - Ordem de chegada.

Mensagens

- Mas o grande problema é o ***tempo de envio***:
  - Garantia e ordem são fornecidos por software
  - A sincronização é complexa porque nunca se sabe se a mensagem já chegou ou não.
    - É possível ter certeza se a mensagem se perdeu ou se atrasou ?

Mensagens

- Primitivas
  - send(id, msg);
  - receive(id, msg);

- Mensagens podem ser “blocantes” ou não;

Produtor/Consumidor com Mensagens

```
producer() {
  msg m, ack;

  while (TRUE) {
    receive(consumer,&ack);
    produce_item(&m);
    send(consumer, &m);
};
```

```
consumer() {
  msg m, ack;

  send(producer, &ack);
  while (TRUE) {
    receive(producer, &m);
    consume_item(&m);
    send(producer, &m);
  };
```

Mensagens e Exclusão Mútua

- Produtor / Consumidor ficou bem mais simples com mensagens
  - Mas e exclusão mútua ?

Porque Utilizar Mensagens ?

- Isolamento dos processos;
- Sem efeitos colaterais
  - Um processo não altera a memória de outro;
- Processos podem desconfiar de outros;
- Permite desenvolvimento independente
  - Basta definir a interface e equipes diferentes desenvolvem baseado na interface;
- Mais geral:
  - Pronto para sistema distribuído!

Variações de Mensagens

- Relação entre mailbox e processo
  - Um mailbox por processo:
    - Nome do processo é usado no send;
  - Sem associação direta
    - Nome do mailbox independe do processo;
    - Mais flexível, mais complexo;
    - Unix

Variações de Mensagens

- Buffering
  - Com: mais eficiente, sender/receiver podem trabalhar em velocidades diferentes;
  - Sem: comunicação síncrona, mais simples e restritivo.

Variações de Mensagens

- Blocantes ou não blocantes
  - Receive blocante: se mailbox estiver vazio, espera até mensagem chegar;
  - Receive não blocante: retorna erro se mailbox estiver vazio;
  - Send blocante: espera até ter espaço no mailbox;
  - Send não blocante: retorna erro se mailbox estiver cheio.

Variações de Mensagens

- Formas adicionais de espera:
  - Vários processos esperam no mesmo mailbox;
  - Um processo espera por vários mailboxes (Unix select)

Mensagems X Memória Compartilhada

- As abordagens utilizando mensagens e dados compartilhados tem o mesmo ***poder de expressão***:
- Resultam, contudo em estilo de programação bem diferentes
- A maioria das pessoas acha o paradigma de dados compartilhados mais fácil de ser usado
- Mas mensagens são mais gerais.

Sincronização no Linux

```
struct wait_queue {
   struct task_struct *task;
   struct wait_queue  *next;
};
void add_wait_queue(queue, entry);
void remove_wait_queue(queue, entry);
```

- Wait queues são modificadas por rotinas de interrupção!!
- Existem wait_queues para todos tipos de evento.

Sleep/Wakeup no Linux

```
void sleep_on(struct wait_queue **queue) {
  struct wait_queue entry = {current, NULL};
  current->state = TASK_UNINTERRUPTABLE;
  add_wait_queue(queue, &entry);
  schedule();
  remove_wait_queue(queue, &entry);
}
void wake_up(struct wait_queue **queue) {
  struct wait_queue *p = *queue;
  do {
    p->task->state = TASK_RUNNING;
    p = p->next;
  } while (p != *queue);
};
```

Semáforos no Linux

```
struct semaphore {
  int count;
  struct wait_queue *wait;
}
void wait(struct semaphore *sem) {
  while (sem->count <= 0)
    sleep_on(sem->wait);
  sem->count--;
}
void signal(struct semaphore *sem) {
  sem->count++;
  wake_up(&sem_wait);
}
```