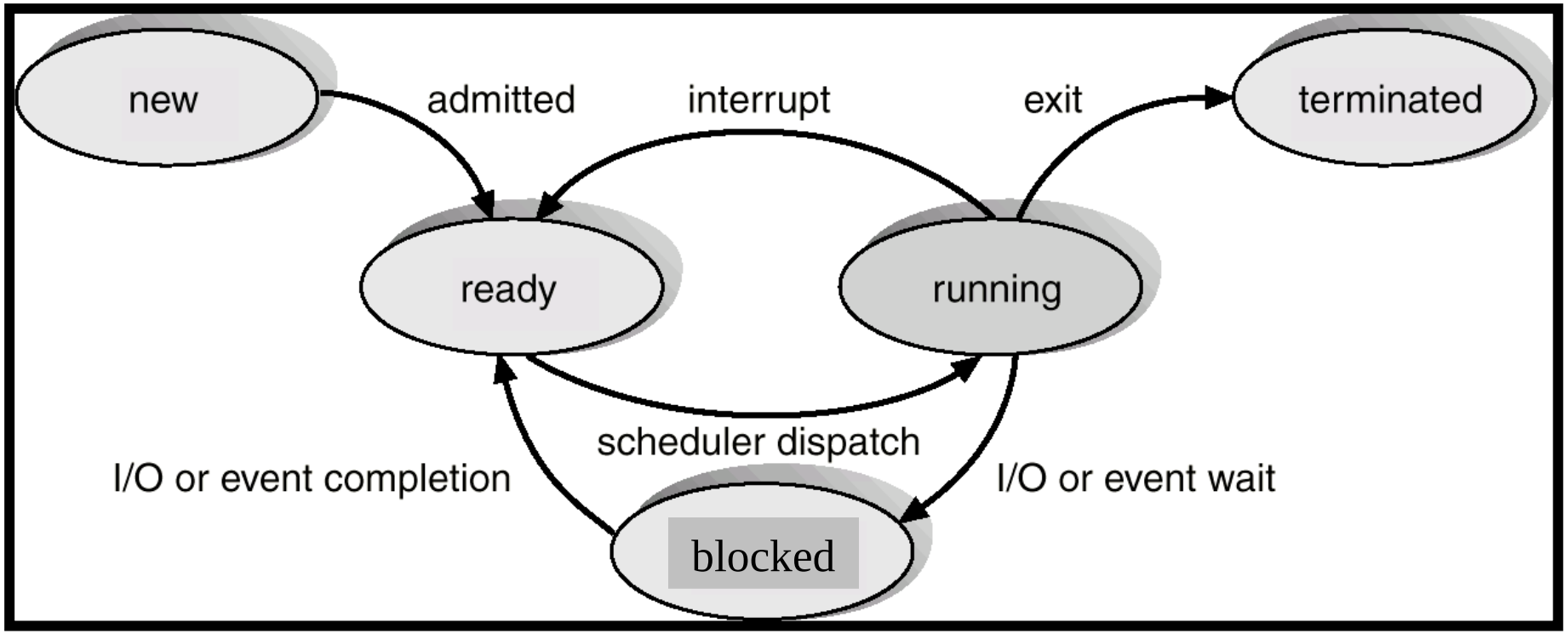
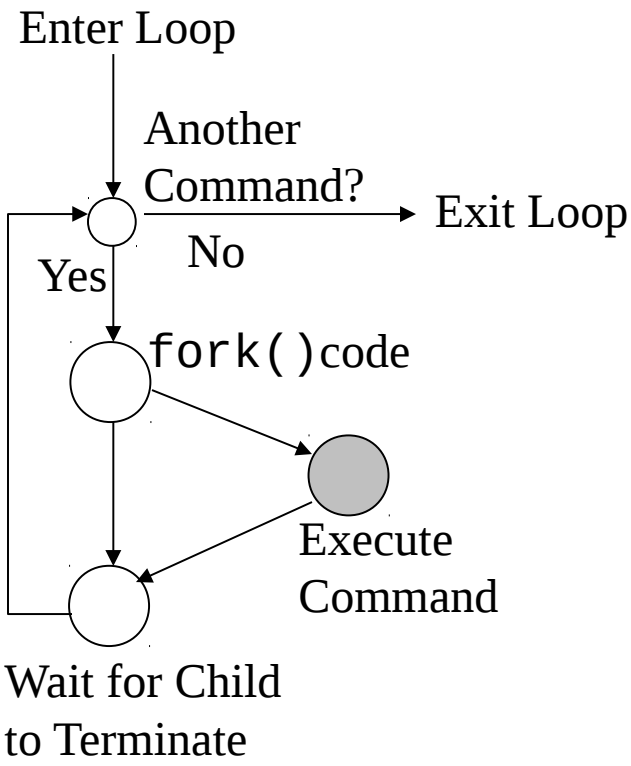
Basic Synchronization Principles



Concurrency

- Value of concurrency – speed & economics
- But few widely-accepted concurrent programming languages (Java is an exception)
- Few concurrent programming paradigm
  - *Each* problem requires careful consideration
  - There is no common model
- OS tools to support concurrency tend to be “low level”

Command Execution



(a) UNIX Shell

Background

- assembly language

x + y

```
mov Ax,x
mov Bx,y
add Bx
```

if (x == y)

```
mov Ax,x
mov Bx,y
sub Bx
jz
```

- a running process can be halted unexpectedly – between two clock cycles, whether or not the execution of an instruction completed
- cannot assume the length, speed, or order of execution
- intermittent bug is next to impossible to replicate
- a bug may not be visible at run-time
- superior programming design on paper
- not adequate to be correct most of the time

critical section problem

example – initial checking balance $800

```
// Teller process
clear check for $500
get balance0
if (balance0 > amount)
      deduct from balance0
save balance0
clear next check
```

```
// ATM process
enter pin
deposit to checking
enter amount $1000
get balance0
add to balance0
save balance0
next transaction
```

Critical Sections (2)

- *Mutual exclusion*: Only one process can be in the critical section at a time
- There is a *race* to execute critical sections
- The sections may be defined by different code in different processes
  - cannot easily detect with static analysis
- Without mutual exclusion, results of multiple execution are not *determinate*
- Programming intervention to ensure correctness

Some Possible Solutions

- Disable interrupts
- Software solution – spin locks
- synchronizing objects
- Transactions
- `FORK()`, `JOIN()`, and `QUIT()` [Chapter 2]
  - Terminate processes with `QUIT()` to synchronize
  - Create processes whenever critical section is complete
- … something new …

Using a Lock Variable

```
shared boolean lock = FALSE;
shared double balance;
```

Code for p_1

```
/* Acquire the lock */
  while(lock) ;
  lock = TRUE;
/* Execute critical sect */
  balance = balance + amount;
/* Release lock */
  lock = FALSE;
```

Code for p_2

```
/* Acquire the lock */
  while(lock) ;
  lock = TRUE;
/* Execute critical sect */
  balance = balance - amount;
/* Release lock */
  lock = FALSE;
```

Solution Constraints

- *Mutual exclusion*: Only one process at a time in the CS
- Only processes competing for a CS are involved in resolving who enters the CS
- Once a process attempts to enter its CS, it cannot be postponed indefinitely
- After requesting entry, only a bounded number of other processes may enter before the requesting process

Semaphore Assumptions

- Memory read/writes are indivisible (simultaneous attempts result in some arbitrary order of access)
- There is no priority among the processes
- Relative speeds of the processes/processors is unknown
- Processes are cyclic and sequential

Semaphores

A *semaphore*, sem, is an integer variable that can only be changed or tested by these two indivisible functions: P (sem) and V (sem)

```
P ( semvar )
        decrement semvar
        if ( semvar is negative )
                block the calling process
                (place in blocked queue)
endP

V ( semvar )
        increment semvar
        if ( semvar queue is not empty )
                resume one process from the queue
endV
```

Notes:
A process executes a P operation to block itself

A running process resumes another process with the V operation

A resumed process may need to wait for its turn to execute

Bounded buffer problem

- shared buffer that can hold 3 items (n = 3)
- producer process puts item (one by one) in an available buffer, i.e., the producer must halt when all 3 spaces are filled
- consumer process takes an available item (one by one, in order) from a buffer, i.e., the consumer must halt when no item is available

- two semaphores: PUT_ITEM = 3, TAKE_ITEM = 0

```
// producer process
loop
        produce an item
        put an item
        increment pointer
endloop
```

```
// consumer process
loop
        take one item from buffer
        increment pointer
        consumer item
endloop
```

P(PUT_ITEM) - This operation determines whether or not there is an empty slot in the buffer memory. If there is an emtpy slot, the producer copies one item to buffer memory. If all slots are full, the OS will block the producer process (moves it to the blocked queue for this semaphore). The semaphore variable PUT_ITEM denotes the state of the buffer - when its value is negative after P decrements PUT_ITEM, all slots are full.

P(TAKE_ITEM) - This operation determines whether or not any slot contains an item to be taken. A negative value of the semaphore variable (as P(TAKE_ITEM) executes) denotes that all slots are empty; therefore, the OS will block the consumer process.

V(TAKE_ITEM) - After one item is copied to buffer memory, the producer needs to alert the consumer process that there is one more item to take (when it gets a chance). If the consumer process happened to be blocked when V(TAKE_ITEM) executes, the OS will resume the consumer process (moves it to ready queue). If the consumer is not blocked, by incrementing the semaphore TAKE_ITEM value, the consumer process will not be blocked when it executes the P(TAKE_ITEM) function.

V(PUT_ITEM) - When one item is taken from buffer memory, the producer process is alerted that there is one more empty slot. The OS will resume the producer process, if it is blocked when V(PUT_ITEM) executes. If it's not blocked, the OS won't block it when the producer executes P(PUT_ITEM) because the value of semaphore PUT_ITEM will not go negative.

Test cases

Some test cases

1) Can the producer process fill the buffer?
2) Can the consumer process empty the whole buffer?
3) Can the producer process put a new item after the buffer is already full?
4) Can the consumer process take an item when the buffer is empty?
5) Any deadlock? Any starvation?

some hints

1. placing P( ) operations: does there exist any instance where a process is required to block before this instruction executes?

2. determining the number of semaphores to use by identifying the requirements/reasons for blocking a process

3. setting initial value of each semaphore: use the largest initial value possible and use zero initial value if a process is required to block immediately (assuming that it is the first to execute)

4. placing V( ) operations: resume a blocked process as soon as possible

critical section example

Initial checking balance $800

```
// TellerA process
clear check for $500
get balance0
if (balance0 > amount)
    deduct from balance0
save balance0
clear next check
```

```
// ATM process
enter pin
deposit to checking
enter amount $1000
get balance0
add to balance0
save balance0
next transaction
```

```
// TellerB process
clear check for $100
get balance0
if (balance0 > amount)
    deduct from balance0
save balance0
clear next check
```

synchronizing

- three concurrent processes
- the first process outputs ‘A’, second outputs ‘B’, third outputs ‘C’
- output pattern: ACBCACBCACBCACBC... (A always first)

Coordination

All processes finish the F functions before starting the G functions.
All functions execute independently.

```
loop repeatedly
    F_one
    G_one
    H_one
endloop
```

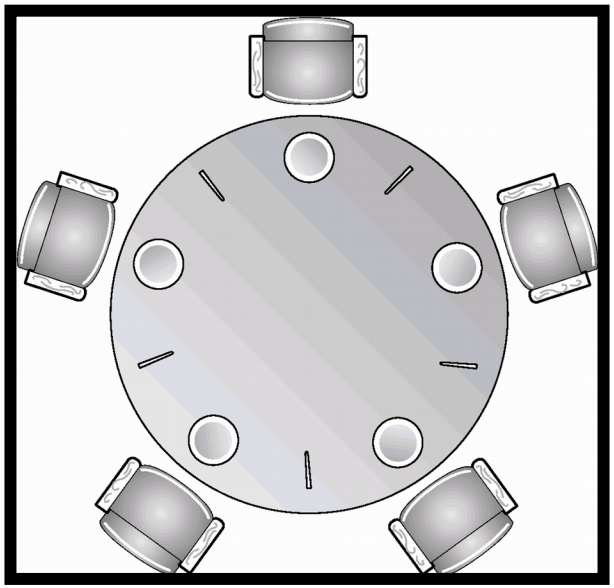
```
loop repeatedly
    F_two
    G_two
    H_two
endloop
```

```
loop repeatedly
    F_three
    G_three
    H_three
endloop
```

Device Controller Synchronization

- The semaphore principle is logically used with the **`busy`** and **done** flags in a controller
- Device driver signals device controller with a **`V(busy)`**, then waits for completion with **`P(done)`**
- Controller waits for work with **`P(busy)`**, then announces completion with **`V(done)`**
- What are the initial values?

Dining-Philosophers Problem



- each philosopher performs each task sequentially and repeatedly
- three tasks: thinking, waiting to eat, eating
- to eat, a philosopher must pick up two chop sticks – on the left and on the right
- chopsticks represents computing resources: disk space, modem, printer, etc.
- maximum currency, no starvation, no deadlock

Dining-Philosophers Problem

THINKING=0, HUNGRY=1, EATING=2,
LEFT=(k-1)%5, RIGHT=(k+1)%5

semaphore mutex // initially 1
semaphore sem[5] // initially 0
int state[5];

take_forks(int k)
    P(mutex)
    state[k]=HUNGRY
    test(k)
    V(mutex)
    P(sem[k])

```
test(int k)
      if (state[k]==HUNGRY &&
                  state[LEFT]!=EATING &&
                  state[RIGHT]!=EATING)
            state[k]=EATING
            V(sem[k])
```

Dining-Philosophers Problem

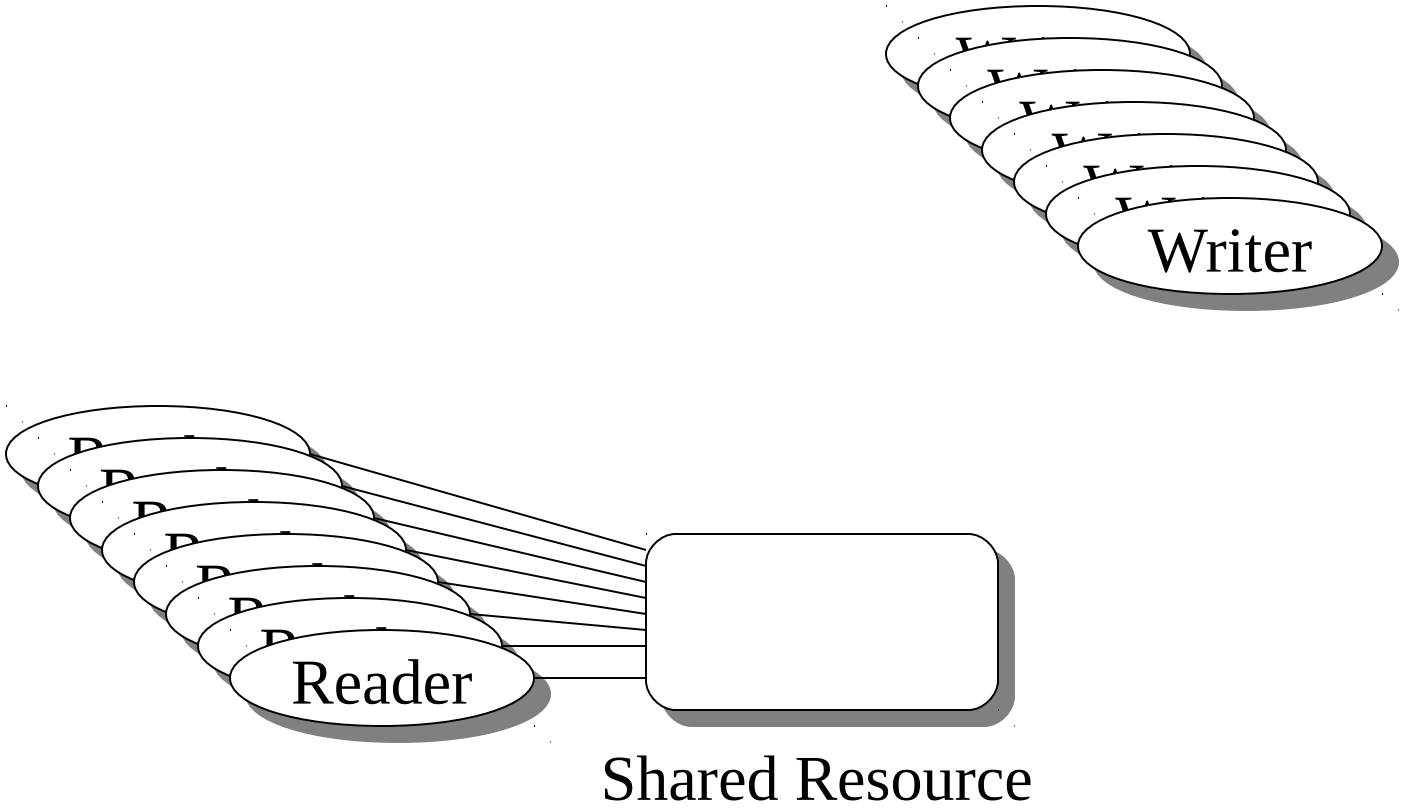
```
put_forks(int k)
    P(mutex)
    state[k]=THINKING
    test(LEFT)
    test(RIGHT)
    V(mutex)

test(int k)
    if (state[k]==HUNGRY &&
                    state[LEFT]!=EATING &&
                    state[RIGHT]!=EATING)
        state[k]=EATING
        V(sem[k])
```

this implementation leaves much to be desired:

1. starvation

2. lacking concurrency

Readers-Writers Problem (3)



Implementing Semaphores

- Minimize effect on the I/O system
- Processes are only blocked on their own critical sections (not critical sections that they should not care about)
- If disabling interrupts, be sure to bound the time they are disabled

Synchronization Hardware

- atomic operation must execute entirely without interruption
- Test and modify the content of a word atomically

```
boolean TestAndSet (donotenter, temp, true) {
    donotenter = temp;
    temp = true;
    return donotenter;
}
```

Active vs Passive Semaphores

- A process can dominate the semaphore
  - Performs V operation, but continues to execute
  - Performs another P operation before releasing the CPU
  - Called a *passive* implementation of V
- *Active* implementation calls scheduler as part of the V operation.
  - Changes semantics of semaphore!
  - Cause people to rethink solutions