2019年10月29日 星期二

Operating System

Lecture 2:

Operating system provide an environment for execution of programs and services to programs and users.

OS services provides:

1User Interface(UI): command line(CLI), GUI, touch-screen, batch

2 Program execution: load the program into memory and run that pro

3 I/O operation

4 File-System manipulation: read and write files, create and delete, search or list and permission

5 Communication: exchange information through process, via shared memory or message passing

6 Error detection: in CPU, hardware or I/O

7 Resource allocation and logging: multiple tasks running concurrently need allocation of memory and logging keep track of which users how much memory

8 protection and security: control the use of information. Protection ensure that all access to resources is controlled. Security is defend invalid access.

Hardware-os-services(上8条)-system calls-interface-program

CLI: implemented in kernel or system program, some implement shells. It fetch command and execute. St commands built in st name of programs

GUI: mouse, keyboard and monitor. Icons represent file program. Various mouse buttons over objects in interface cause various actions.

System calls:

1 programming interface to service provides by os

2 written in high level language

3 accessed by programs via API(application programming interface) rather than direct system call use

A number associated with each system call and system call interface maintains table indexed according to these numbers. System call interface invoke intended system call in OS kernel and returns any value and status of system call.

System call parameter passing:

3 methods:

1 pass in registers. 2 stored in block, or table in memory 3 parameters placed onto stack by pro and popped off by os

TYPES OF SYSTEM CALLS:

Process control: create or terminate process. End or abort. Get or set process attributes. Load or execute. Wait time. Signal event. Allocate or free memory. Return error and debug. Locks for managing access to shared data between processes.

File: create/delete/write/read/get/set

Device manage: request device or release. read/write/reposition. get/set attributes.attach or detach devices.

Information maintenance: get/set time date, system data, process, file, device attributes

Communication: create/delete.send message, gain access to memory. Attach or detach remote devices.

Protection: get/set permission and allow or deny access

SYSTEM SERVICES:

File management

Status information: date time number of users available memory

Programming-language support compilers, assemblers, debuggers and interpreters

Program loading and execution

Communications:provide mechanism for creating virtual connections among processes users and computer systems

Background services:provide facilities like disk checking, process scheduling, error logging, printing

Linkers and Loaders:

Compiled code — object files(relocatable) linker combines them into one executable file— brought by loader into memory to execute(to final addresses)

Apps compiled on one system usually not executable on other os, each os provides own system calls. If written in standard language or high Lebel languages that can be interpreted in multiple os can be multi-operating

ABI is equivalent to API defines how different components of binary code can interface for a given operating system on a given architecture

OS DESIGN:

User goal: operating system should be convenient to use, easy to learn, reliable, safe and fast

System goal: OS should be easy to design, implement and maintain

Operating System Structure:

Monolithic Structure — Original Unix

Contains two part: System programs , kernel(everything below system-call interface and above physical hardware, file system, cpu, memory, functions)

Hardware-device drivers-kernel-system-call interface-application

Microkernel structure: communications take place between user modules using message passing

Benefits:

1 easier to extend a microkernel

2 easier to port the os to new architecture

3 more reliable and secure(less code running in kernel mode)
microkernel implement user services and kernel services in different address spaces while monolithic one implement both user services and kernel services under same address space

The size of monolithic kernel is larger since both kernel services and user services reside in the same address
The execution of monolithic kernel is faster as communication between application and hardware devices depend on system call whereas the communications are established through message passing in microkernel

2019年10月29日 星期二

Kernel Mode
In Kernel mode, the executing code has complete and unrestricted access to the underlying hardware. It can execute any CPU instruction and reference any memory address. Kernel mode is generally reserved for the lowest-level, most trusted functions of the operating system. Crashes in kernel mode are catastrophic; they will halt the entire PC.

User Mode
In User mode, the executing code has no ability to directly access hardware or reference memory. Code running in user mode must delegate to system APIs to access hardware or memory. Due to the protection afforded by this sort of isolation, crashes in user mode are always recoverable. Most of the code running on your computer will execute in user mode.

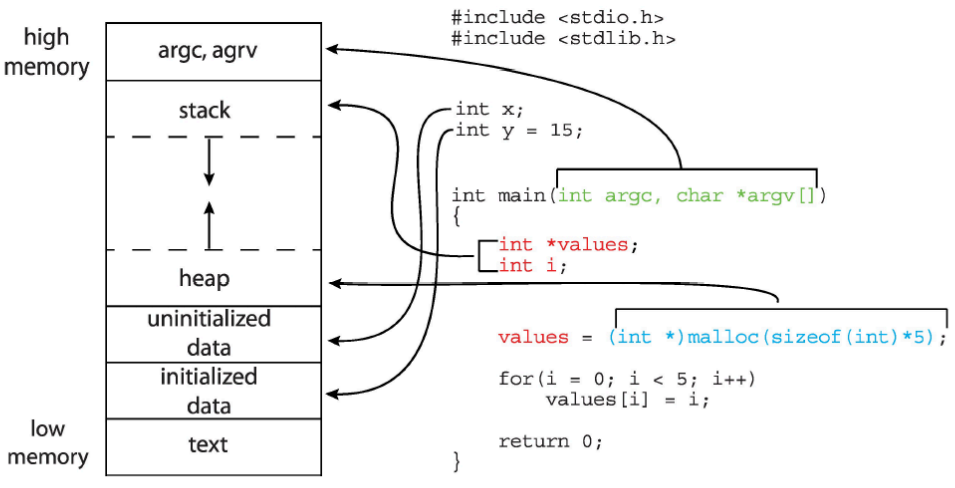
loadable kernel modules (LKMs)
- Uses object-oriented approach
- Each core component is separate
- Each talks to the others over known interfaces
- Each is loadable as needed within the kernel

Most modern operating systems are actually not one pure model
Lecture 3

Process – a program in execution; process execution must progress in sequential fashion.
Contains: program code-text section
Program counter
stack(function para, address, local variable)
Data section(containing global variables)
Uninitialized data Is called bss(block started by symbol)
heap(containing memory dynamically allocated during run time)

Program is passive entity stored on disk whereas process is active

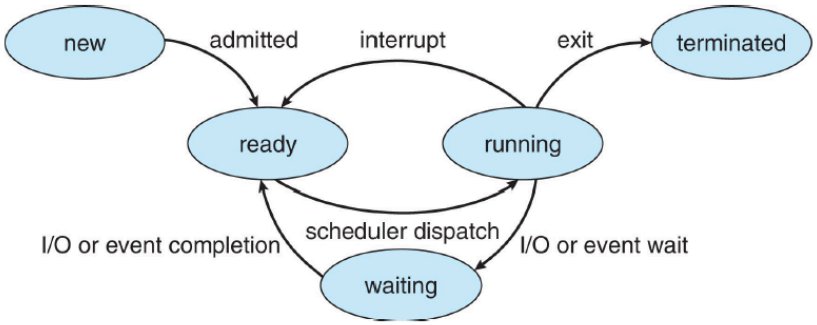
Memory Layout of a C Program



That is program becomes process when executable file loaded into memory
Execution started via GUI mouse click and command line entry
One program can be several process

Stack 往下走 heap往上来 紧跟和data text

Process state:
NEW: created
Running :executed
Waiting :wait for sth
Ready :waiting to be assigned to processor
Terminated: end



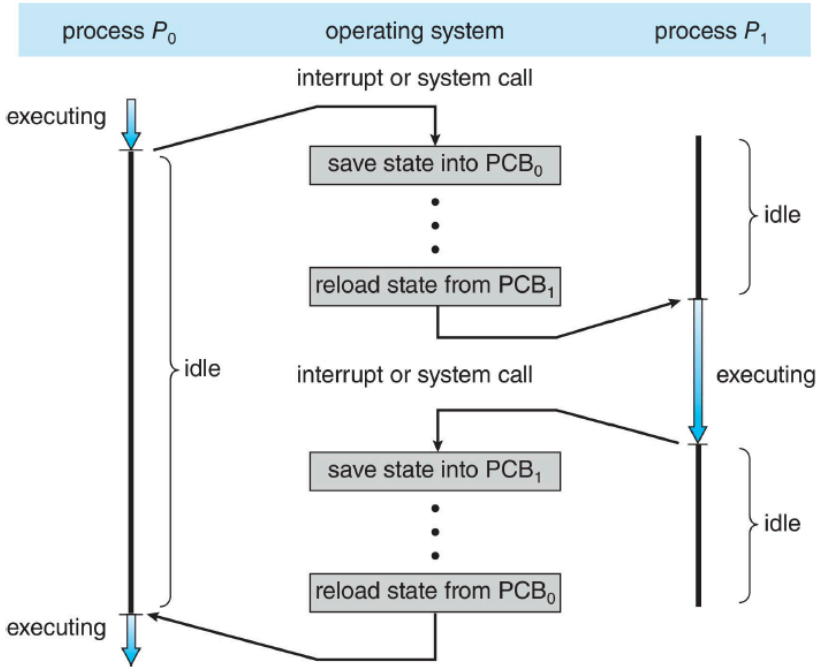
PCB: process control block
Process state
Program counter: location of instructions to next excuse
Cpu register and scheduling information: priority
Memory management: allocations
Accounting information and i/o status: cpu used and io devices

Threads:
Process scheduling:
Maximize cpu use, quickly switch processes onto cpu core
Process scheduler selects among available processes for next execution on cpu core
Two scheduling queues:
Ready queue: set of all processes residing in main memory, ready and waiting to execute

Wait queue: set of processes waiting for an event

CPU switch from process to process:

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



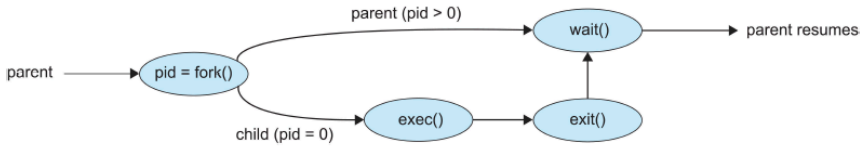
When cpu switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch
Context of a process represented in the pcb
And context switch time is overhead

Multitasking in mobile systems:
Some mobile systems allows only one process to run others suspended.
IOS: single foreground process- controlled via user interface
Multiple background processes in memory, running but not on the display, with limits

Android: runs foreground and background with fewer limites
Background process uses a service to perform tasks
Services can keep running even if background process is suspended and has no user interface, with small memory use

Operations on process:
System must provide mechanisms for process creation and process termination
Process creation: parent create child which in turn create other—-tree process

- 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()` for the child to terminate



Process identified and managed via process identifier(pid)
Sharing-resource option: share all, child share subset of parent, share no
Execution: concurrently, parent wait for child

Process termination:
Process executes last statement and then asks the operating system to delete it using the exit() sys call:
Return status data from child to parent(via wait)
Process resources are deallocated by operating system

Parents may terminate the execution of children processes using the abort sys call:
Reasons:
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

Cascading termination: all children are terminated
If no parent waiting (did not invoke `wait()`) process is a zombie
If parent terminated without invoking `wait` , process is an orphan

Terminate process from most to least important:n android terminate from least one
Foreground process
Visible process
Service process
Background process
Empty process

Interprocess communication:
Processes within a system maybe independent or cooperating
Cooperating process can affect or be affected by other process including sharing data. Using it for information sharing, modularity and convenience

Two models of IPC: shared memory and message passing

Shared-memory: communication is under control of users not os
Major issue is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory

Producer-consumer problem: consumer consumes the production by producer

Unbounded-buffer places no practical limit on the size of the buffer
Bounded-buffer assumes that there is a fixed buffer size

Message passing: mechanisms for processes to communicate and to synchronize their actions
IPC facility provides two operations: send and receive message
Need to establish a communication link:
Physical: shared memory, hardware and network
Logical: direct and indirect, synchronous or asynchronous, Automatica or explicit buffering

Direct communication:
send(p, message) send to p
receive(q, message) receive from q
Properties of communication link:
Links are established automatically
Link is associated with exactly one pair of communicating processes
Between each pair there exists exactly one link
Link should be bidirectional but st unidirectional

Indirect communication:
Messages are directed and received from mailboxes:
Each mail box has a unique id and processes can communicate only if they share m mailbox

Properties of communication link:
Built link if processes share common mailbox
A link may be associated with many process
Each pair of process may share several communication links
Link may be unidirectional or bidirectional

operations:
Create a new mailbox(port)
Send and receive message through mailbox

Destroy a mailbox

Synchronization:
Blocking is considered synchronous
Blocking send: the sender is blocked until the message is received
Blocking receive: the receive is blocked until a message is available

Non-blocking is considered asynchronous
Non-blocking send: the sender sends the message and continue
Non-blocking receive: the receiver receives A valid message or null message

If both send and receive are blocking, we have a rendezvous

Buffering:
Queue of messages attached to the link
Implemented in one of three ways
1 zero capacity— no messages are queued on a link. Sender must wait for receiver(rendezvous)
2 bounded capacity — finite length of n messages. Sender must wait if link full
3 Unbounded capacity — infinite length. Sender never waits

Examples of IPC Systems —- posix
Pipes
Ordinary pipes: cannot be accessed from outside the process that created it.
Typically, parent process creates a pipe and uses it to communicate with a child process that it created.
It's in standard producer-consumer style write in one end and read in another unidirectional.
Named pipes: can be accessed without a parent-child relationship
Communication is bidirectional several processes can use the named pipe for communication

Lecture 4 processes and input/output

Each process has an array of handles which correspond to an external device
We refer handles as file descriptors：
A array structure maintained by the kernel for each process
Held in the kernel memory and process can modify them through syscall

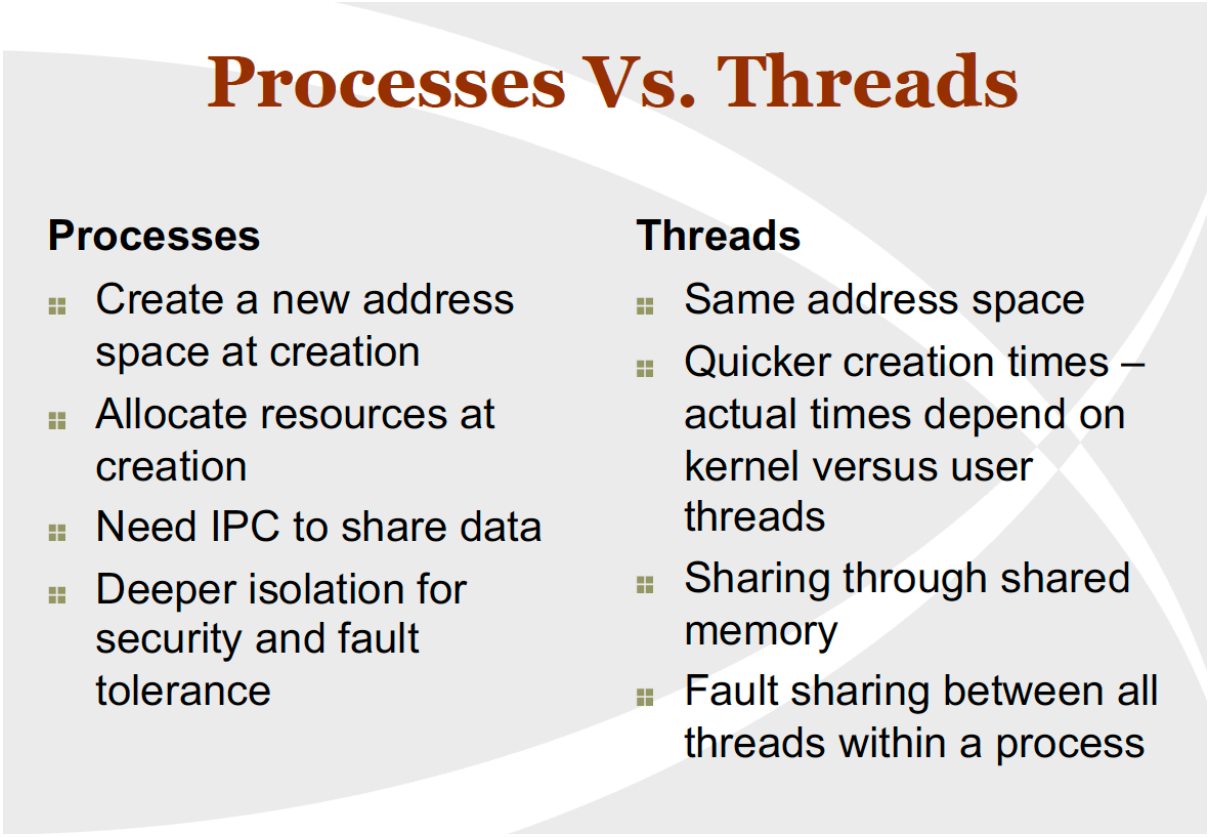
Int fd = open(filename)
The value returned by open is file descriptor pointing to the given file

Output redirection: close 1 and open text afterwards

Command piping: connect the output of a process to the input of another process
1: use the pipe() system call and create a pipe. Allows unidirectional data flow with write end and read end

2: fork the process
3: rewrite the file descriptor

Lecture5 Threads
Threads run within a process
Multiple tasks within the application can be implemented by separate threads
Process creation is heavy-weight while thread creation is light-weight
Can simplify code and increase efficiency
Kernels are generally multithreaded



Multithreaded benefits
Responsiveness: may allow continued execution if part of process is blocked, especially important for user interfaces
Resource sharing:threads share resources of process, easier than shared memory or message passing
Economy: chapter than process creation, thread switching lower overhead than context switching
Scalability: process can take advantage of multiprocessor architectures

Multicore programming:
Multicore or multiprocessor
Parallelism: A system can perform more than one task simultaneously
Concurrency:supports more than one task making progress. Note single processor/ core provides concurrency

Parallelism belongs to concurrency.

Types of parallelism:
Data parallelism: distributes subsets of the same data across multiple cores, same operation on each(切data给不同core)
Task parallelism: distributing threads across cores, each thread performing unique operation(完整data同时给不同core)

Amdahl's law
S is serial portion N processing cores
Speedup <= 1/(s+(1-s)/n)

User and kernel threads
User threads: management done by user-level threads library

Kernel threads: support by the kernel
Posix pthreads is a thread programming standard most of the time implemented as kernel-level threads

Multithreading models:
Many to one
Many user-level threads mapped to single kernel thread
One thread blocking causes all to block
Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time

One to one
Each user-level thread maps to kernel thread
Creating a user-level thread Cretes a kernel thread
More concurrency than many-to one
Number of threads per process sometimes restricted due to overhead

Many to many
Allows many user level threads to be mapped to many kernel threads
Allows the os to create a sufficient number of kernel threads

Thread libraries
It provides programmer with API for creating and managing threads
Two ways of implementing: library entirely in user space and kernel-level library supported by the OS

There are four threads executing in linux
Pthreads: may be provided wither as user-level or kernel-level
Pthread creation: process has the main thread at the beginning
New thread continues with start and main continues with the statement after
Pthread termination:

The thread terminates for the following:
The Strat function performs a return
Threads calls a pthread-exit function
Threads is cancelled using pthread-cancel
Main thread returns

Thread can obtain own id using pthread-self
If two threads id are same use pthread-equal function

Join a terminated thread
A thread can wait for another thread using the pthread-join function
If a created thread is not detached we must join with it or it will create a zombie thread

Detaching a thread
If no thread is interested in joining we need to detach the thread
Pthread-detach
It is not possible to join to a detached thread

Thread attributes:
Attributes can be used to set properties of threads- such as detached

Protecting shared variables:
Advantage of threads: can share via global variables
Must ensure multiple threads are not modifying the variables at the same time
Use a pthread-mutex variable(lock and unlock)
If creating two threads to modify global variable, we use lock after one thread reached the loop

Thread cancellation:
Terminating thread before it has finished:
Two approach: 1 asynchronous cancellation, terminates the target immediately
2 deferred cancellation allows the target thread to periodically check if it should be cancelled

Actual cancellation depend on thread sated what for it enable cancellation

Lecture 6 synchronization

Concurrent process:
It can compete for shared resources and cooperate with each other in sharing the global resources

OS deals with competing processes:
Carefully allocating resources
Properly isolating processes from each other

It also deals with cooperating processes by providing mechanisms to share resources

Competing
Processes that do not work together cannot affect the execution of each other but they can compete for devices and other resources

Properties: can stop and restart without side effects and can proceed at arbitrary rate

Cooperating
Processes that are aware of each other and directly/in work together may affect the execution of each other

Ex transaction processes in an airline reservation system
Property share common object and exchange information. Non-deterministic and subject to race condition

Benefits: we can share resources and do thing faster like concurrently

Race condition:
When more than two processes are reading or writing shared data and final results depend on who runs precisely when is a race condition

Avoid: prohibit more than one process from reading. And writing shared data at same time
Essentially we need mutual exclusion

Mutual exclusion:
When one process is reading or writing a shared data other processes should be prevented from doing the same
Providing mutual exclusion in os is a major design issue

Critical section:
Part of the program that accesses shared data
If we arrange the program uns such that no two processes are in their critical sections at the same time ,race conditions can be avoided

Requirement for critical section
No two processes may be simultaneously in their critical section
No assumptions be made about speeds or number of cpus
No process running outside critical section may block other processes
No process should have to wait forever to enter its critical section

Critical sections execution should be atomic running all or none
Should not be interrupted in the middle and for efficiency we need to minimize length of critical section

2019年10月29日 星期二

Road to solution:
Use lock variables to prevent two processes entering the critical section at the same time- all along we are taking about a single critical section
If lock=0, set lock=1 and enter, ow, wait

Strict alternation: two processes take turns in entering the critical section
It can cause starvation
Other drawbacks: busy waiting cause by continuously waiting for a value of a variable, waste cpu time and the lock we used in busy waiting is called spin lock
Critical section should be tiny here

Mutual exclusion:

Mutual Exclusion: Firs Attempt

- The simplest mutual exclusion strate taking turns as considered earlier

```
/* process 0 */
.
.
while (turn != 0);
/* critical section */
turn = 1;
.
.
```

```
/* process 1 */
.
.
while (turn != 1);
/* critical section */
turn = 0;
.
.
```

Use value variable turn

Mutual Exclusion: Third Attempt

- This works, i.e., provides mutual exclusic
- Has **deadlock** – why?

```
/* process 0 */
.
.
flag[0] = true;
while (flag[1]);
/* critical section */
flag[0] = false;
.
.
```

```
/* process 1 */
.
.
flag[1] = true;
while (flag[0]);
/* critical section */
flag[1] = false;
.
.
```

Mutual Exclusion: Fourth Attempt

- This works
- Has **livelock** – why?

```
/* process 0 */
.
.
flag[0] = true;
while (flag[1])
{
        flag[0] = false;
        /* random delay */
        flag[0] = true;
}
/* critical section */
flag[0] = false;
.
.
```

```
/* process 1 */
.
.
flag[1] = true;
while (flag[0])
{
        flag[1] = false;
        /* random delay */
        flag[1] = true;
}
/* critical section */
flag[1] = false;
.
.
```

Peterson's Algorithm

```
/* process 0 */
...
flag[0] = true;
turn = 1;
while (flag[1] &&
        turn == 1);
/ * critical section */
flag[0] = false;
/* remainder */
...
```

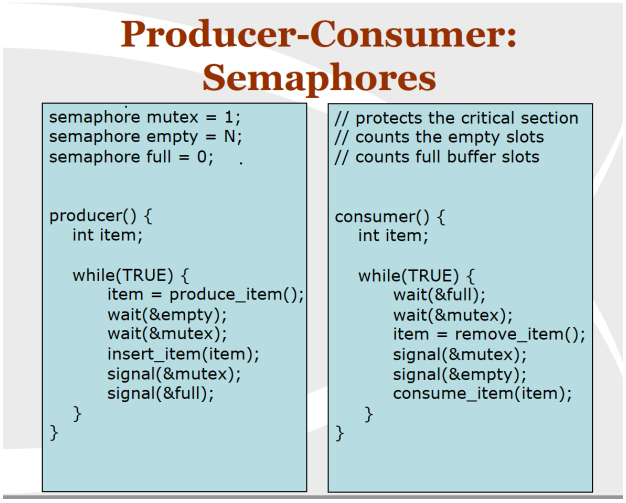
```
/* process 1 */
...
flag[1] = true;
turn = 0;
while (flag[0] &&
        turn == 0);
/* critical section */
flag[1] = false;
/* remainder */
...
```

Property of machine instruction approach
Advantages:
applicable to any number of processes
can be used with single processor or multiple processors that share a single memory
simple and easy to verify
can be used to support multiple critical sections, i.e., define a separate variable for each critical section

Disadvantages:
Busy waiting is employed – process waiting to get into a critical section consumes CPU time
Starvation is possible – selection of entering process is arbitrary when multiple processes are contending to enter

Semaphore:
to transmit a message via a semaphore a process executes signal(s)
to receive a message via a semaphore a process executes wait(s)

Operations defined on a semaphore:
can be initialized to a nonnegative value – set semaphore
wait – decrements the semaphore value – if value becomes negative, process executing, wait is blocked
signal – increments the semaphore value – if value is not positive, a process blocked by a wait operation is unblocked



A monitor ensures that only one process at a time can be active within the monitor – so you don't need to code this explicitly

Problems with synch Primitives
Starvation: the situation in which some processes are making progress toward completion but some others are locked out of the resource(s)
Deadlock: the situation in which two or more processes are locked out of the resource(s) that are held by each other

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- Pseudocode syntax of a monitor:

```
monitor monitor-name
{
  // shared variable declarations
  function P1 (…) { …. }

  function P2 (…) { …. }

  function Pn (…) {……}

  initialization code (…) { … }
}
```

Condition Variables

- `condition x, y;`
- Two operations are allowed on a condition variable:
  - `x.wait()` – a process that invokes the operation is suspended until `x.signal()`
  - `x.signal()` – resumes one of processes (if any) that invoked `x.wait()`
    - If no `x.wait()` on the variable, then it has no effect on the variable

Classification of resources
Reusable and consumable
The one that can be safely used by one process at a time and is not depleted bu that use
These can be created and destroyed

2 preempt able: can be taken away from the process owing it with no ill effects
Non preempt able: cannot be taken away from the process from current Owen without causing computation fail

Conditions for deadlock:
Mutual exclusion
Hold and wait
No preemption
Circular wait