**6.2 The Critical-Section Problem**

Critical Section:

section of code that is responsible for changing data and must only be executed by one thread or process at a time to avoid a race condition

accessing/updating data that is shared with at least one other process

no two processes are executing their critical sections at the same time

*Critical-section Problem:*

synchronize process activity so they can cooperatively share data

Each process must:

* request permission to enter critical section (entry section)
* critical section
* followed by exit section
* any remaining code is the remainder section

while (true) {

entry section

critical section

exit section

remainder section

}

A solution to this problem must include these requirements:

1. Mutual Exclusion
   1. Only one process at a time may execute its critical section
2. Progress
   1. Progress must be made if there are process waiting to enter their critical sections and cannot be postponed forever
3. Bounded waiting
   1. A process cannot wait forever while others enter their critical sections, it cannot be pushed aside. This bound or limit is the number of times other processes are allowed to enter their critical sections after a process has requested to do the same before that request is granted

\*\*At any time the operating system kernel code is subject to several possible race conditions as there could be any number of kernel-mode processes at any time.

if fork() were called by 2 different processes without mutual exclusion then the same process identifier # could be provided to both children

Two Approaches to handle critical sections in OS:

1. Preemptive kernels
   1. more responsive, since less risk that a kernel-mode process will run for a very long time
2. Nonpreemptive kernels
   1. does not allow a process running in kernel mode to be preempted, a kernel-mode process runs until it exits, blocks, or voluntarily yields control of CPU
   2. essentially free from race conditions on kernel data structures, since only one process is active in kernel at a time

**6.3 Peterson’s Solution**

Software based solution to critical section problem

\*\*May not work correctly on modern computer architectures because of how they perform operations such as load and store

2 process that alternate execution between critical section and remainder section (P0 and P1)

int turn;

boolean flag[2];

turn indicates whose turn it is to enter the critical section, if turn == 1 the the process is allowed to execute it’s critical section.

while (true) {

flag[i] = true;

turn = j;

while (flag[j] && turn == j);

// critical section

flag[i] = false;

//remainder section

}

In Peterson's solution, the \_\_\_\_ variable indicates if a process is ready to enter its critical section.

A: flag[i]

**6.5 Mutex Locks**

Mutex lock: mutual exclusion lock, simple software for assuring mutual exclusion

used to protect critcal sections and prevent race conditions

Process must acquire the lock before entering a critical section and releases the lock when it exits the critical section.

acquire()

release()

while(true){

acquire lock

critical section

release lock

remainder section

}

available – boolean variable that indicates whether a lock is available or not

if it is available: acquire() is called and lock is considered unavailable

acquire() {

while(!available);

//busy wait

available = false;

}

release() {

available = true;

}

high contention: large number of threads attempting to acquire the lock

decrease overall performance of concurrent applications

low contention: low number of threads attempting to acquire the lock

spinlocks; locking mechanism of choice on multiprocessor systems when lock is to be held for a short duration

short duration = less than 2 context switches

advantage: no context switch is required when a process must wait on a lock

busy waiting: thread or process using CPU time continuously while waiting for something

any process that tries to enter its critical section must loop continuously in call to acquire() if a process is already in its critical section

**6.6 Semaphores**

semaphore: integer variable accessed through wait() and signal()

wait(S) {

while(S <= 0);

//busy wait

S--;

}

signal(S) {

S++;

}

binary semaphores are similar to mutex locks

counting semaphores: range over unrestricted domain

Mutual Exclusion: at least one resource must be held in a non sharable mode, only one thread/process may use a resource at a time

Hold & Wait: a thread must be holding at least one reource and waiting to acquire additional resources that are currently being held by other threads

No preemption: Resources cannot be preempted, resources can be released only voluntarily by the thread

Circular Wait: there exists thread T0 T1 and T2 such that T0 is waiting for a resource held by T1 which is waiting for a resource held by T2 which is waiting for a resource held by T0.