Chapter 11.6: Thread Synchronization

Consistent View

* The variables each thread use is exclusive from each other
* A variable is read-only but used by multiple threads

Inconsistencies when reading variables

* Interleaving memory cycles
* Use a lock to allow access one variable at a time

Increment Operation:

1. Read memory location
2. Increment value
3. Write new value

11.6.1) Mutexes

A mutex is a lock that is set before accessing a shared resource and then released when done. Any other thread will be blocked while mutex is set until it is released.

If multiple threads are blocked when mutex is released, then the first thread to run will be able to set the new lock. The others will continue to be locked and have to wait for their turn (mutual exclusion mechanism)

11.6.2) Deadlock Avoidance

Controlling the order of mutexes locking. Release locks and try again or use pthread\_mutex\_trylock.

11.6.3) pthread\_mutex\_timedlock function

Function allows binding the time a thread uses to block when a mutex it is trying to acquire is already locked.

11.6.4) Reader-writer locks

Similar to mutexes but allows a higher degree of parallelism

States: locked in read mode, locked in write mode, and unlocked

11.6.5) reader-writer locking with timeouts

11.6.6) Condition variables

Another synchronization mechanism available to threads. Provides a place for threads to rendezvous. Allows threads to wait in a race-free way for arbitrary conditions to occur when used with mutexes. Condition protected by a mutex, because a thread can only modify the condition if the mutex is locked.

11.6.7) Spin locks

It is basically a mutex, but instead of blocking a process by sleeping, it is blocked by busy waiting (spinning) until lock can be acquired. Commonly used in situations where locks are held for short periods of times and threads don’t want to incur the cost of being descheduled.

11.6.8) Barriers

Another form of synchronization mechanism that can be used to coordinate multiple threads working in parallel. It allows each thread to wait until all cooperating threads have reached the same point, and then continues execution from there.

Chapter 15.2: Pipes

Pipes are the oldes form of UNIX System IPC and are provided by all UNIX systems.

Two limitations:

1. Data only flows in one direction
2. Can be used only between processes that have a common ancestor

Int pipe (int fd[2]);

* Creates a pipe
* Returns 0 if ok and -1 on error
* Two file descriptors are returned through the fd argument
  + Fd[0] = open for reading
  + Fd[1] = open for writing
  + Output of fd[1] is the input for fd[0]
* Fstat function (section 4.2) returns file type of FIFO for file descriptor of either end of a pipe
* For a pipe from the parent to the child, the parent closes the read end of the pipe (fd[0]) and the child closes the write end (fd[1])
* Switch around for a pipe from child to the parent

If one end of a pipe is closed:

1. Write end is closed, read returns 0 to indicate EOF
2. Read end is closed, signal SIFPIPE is generated, write returns -1 or errno is set to EPIPE

Writing to a pipe, constant PIPE\_BUF specifies the kernel’s pipe buffer size.

Write of PIPE\_BUF bytes or less will not be interleaved with the writes from other process to the same pipe, but if more than PIPE\_BUF bytes than it will.

PIPE\_BUF bytes determined using pathconf/fpathfonf

Chapter 15.3) popen and pclose functions

Handles creating a pipe, forking a child, closing the unused ends of the pipe, executing a shell to run the command, and waiting for the command to terminate.

FILE \*popen (const char \*cmdstring, const char \*type):

* Does a fork and exec to execute the cmdstring and returns a standard I/O file pointer
* If type is “r”, the file pointer is connected to the standard output of cmdstring
* If type is “w”, the file pointer is connected to the standard input of cmdstring

Pclose

* Closes the standard I/O stream, waits for the command to terminate, and returns the termination status of the shell
* If shell cannot be executed then, exit(127)

Chapter 15.8: Semaphores

A semaphore is a counter used to provide access to a shared data object for multiple processes.

To obtain a shared resource, a process needs:

1. Test the semaphore
2. If > 0, process can use resource. Value will decrement by 1, indicating that process used one unit of resource
3. == 0, process goes to sleep until semaphore value > 0. Returns to step 1 when awake

Semaphore value incremented by 1 when process is done using shared resource, and other waiting processes are woken up.

Steps 1 and 2 must be an atomic operation. Semaphore implemented in kernel.

Binary semaphore controls a single resource and its value is initialized to 1.

Semaphores can be initialized to any positive value, it represents how many units of the shared resource are available.

XSI semaphores:

1. Not simply a single non-negative value. A set of one or more semaphore values. When semaphore is created, we need to specify number of values in the set
2. Creation of a semaphore (semget) is independent of its initialization (semctl). This is a fatal flaw, since a new semaphore set cannot be initialized and created atomically.
3. All forms of XSI IPC remain in existence even when no process is using them. Program that terminates without releasing the allocated semaphores. Handled by undo feature.

Kernel maintains a semid\_ds structure for each semaphore set, each semaphore is represented by an anonymous structure.

Int semget(key\_t key, int nsems, int flag)

* Returns semaphore ID If ok, -1 on error
* Key is converted into an identifier
* Nsems = number of semaphores in the set. If creating new set, nsems must be specified, if referencing a set, then specify nsems = 0

Int semctl(int semid, int semnum, int cmd, …/\* union semun arg \*/)

* Fourth argument is optional
* Cmd specifies the commands to be performed on the set specified by semid

Int semop(int semid, struct sembuf semoparray[], size\_t nops)

* Semoparray argument is a pointer to an array of semaphore operations, represented by sembuf structures
* Nops arguments specifies the number of operations in the array, the operation on each member of the set is specified by the correspoinding sem\_op value, which can be negative, 0, or positive.
  + Positive: returning of resources by the process, value added to the semaphore’s value, if undo flag is specified, sem\_op is also subtracted from the semaphore’s adjustment value
  + Negative: obtain resources that the semaphore controls
  + 0: calling process wants to wait until the semaphore value becomes 0