Home Assignment -2

[1]

Answer:

* RUN to READY can be caused by a time-quantum expiration
* READY to NONRESIDENT occurs if memory is overcommitted, and a process is
* temporarily swapped out of memory
* READY to RUN occurs only if a process is allocated the CPU by the dispatcher
* RUN to BLOCKED can occur if a process issues an I/O or other kernel request.
* BLOCKED to READY occurs if the awaited event completes (perhaps I/O
* completion)
* BLOCKED to NONRESIDENT - same as READY to NONRESIDENT.

[2]

Answer:

At time 22

P1: blocked for I/O

P3: blocked for I/O

P5: ready/running

P7: blocked for I/O

P8: ready/running

At time 37

P1: ready/running

P3: ready/running

P5: blocked suspend

P7: blocked for I/O

P8: ready/running

At time 47

P1: ready/running

P3: ready/running

P5: ready suspend

P7: blocked for I/O

P8: exit

[3]

0

<child pid>

or

<child pid>

0

[4] List reasons why a mode switch between threads may be cheaper than a mode switch between processes.

Answer:

* Switching process requires OS to process more information.
* Memory is shared by threads, so there's no need to exchange memory or data during thread creation or switching.
* Thread switching does not require kernel to get involved, which in turn saves time on switching user to kernel mode.

[5] List three advantages of ULTs over KLTs.

Answer:

1. Thread switching does not require kernel mode privileges because all of the

thread management data structures are within the user address space of a single

process. Therefore, the process does not switch to the kernel mode to do thread

management. This saves the overhead of two mode switches (user to kernel; kernel

back to user).

2. Scheduling can be application specific. One application may benefit

most from a simple round-robin scheduling algorithm, while another might benefit

from a priority-based scheduling algorithm. The scheduling algorithm can be

tailored to the application without disturbing the underlying OS scheduler.

3. ULTs can run on any operating system. No changes are required to the underlying kernel to support ULTs. The threads library is a set of application-level utilities shared by

all applications.

[6] List two disadvantages of ULTs compared to KLTs.

Answer:

1. In a typical operating system, many system calls are blocking. Thus, when a ULT

executes a system call, not only is that thread blocked, but also all of the threads

within the process are blocked.

2. In a pure ULT strategy, a multithreaded

application cannot take advantage of multiprocessing. A kernel assigns one process

to only one processor at a time. Therefore, only a single thread within a process can

execute at a time.

[7] In the discussion of ULTs versus KLTs, it was pointed out that a disadvantage of ULTs is that when a ULT executes a system call, not only is that thread blocked, but also all of the threads within the process are blocked. Why is that so?

Answer:

Because, with ULTs, the thread structure of a process is not visible to the operating

system, which only schedules on the basis of processes.

[8] Consider an environment in which there is a one-to-one mapping between user-level

threads and kernel-level threads that allow one or more threads within a process to issue blocking system calls while other threads continue to run. Explain why this model can make multithreaded programs run faster than their single-threaded counterparts on a uniprocessor computer.

Answer:

The issue here is that a machine spends a considerable amount of its waking hours

waiting for I/O to complete. In a multithreaded program, one KLT can make the

blocking system call, while the other KLTs can continue to run. On uniprocessor, a

process that would otherwise have to block for all these calls can continue to run its

other threads.

[9] If a process exits and there are still threads of that process running, will they continue to run?

Answer:

No. When a process exits, it takes everything with it—the KLTs, the process

structure, the memory space, everything—including threads.

[10] What is the distinction between competing processes and cooperating processes?

Answer:

Competing processes - compete for resources. For example, two independent applications may both want to access the same disk or file or printer. The OS must regulate these accesses.

Cooperating Processes - Share resources. May or may not be aware of each other. Some processes are designed to cooperate together (jointly) on the same activity and share resources. They may also be aware of each other by process id.

[11] What is the difference between strong and weak semaphores?

Answer:

Strong semaphore specifies in which order processes are removed from the waiting queue (such as FIFO) Weak semaphore does not specify this.

[12] What is a monitor?

Answer:

The monitor is a programming-language construct that provides equivalent functionality to that of semaphores and that is easier to control.

[13] What is the distinction between *blocking* and non blocking with respect to messages?

Answer:

* Blocking send, blocking receive: Both the sender and receiver are blocked until the message is delivered; this is sometimes referred to as a rendezvous. This combination allows for tight synchronization between processes.
* Nonblocking send, blocking receives: Although the sender may continue on, the receiver is blocked until the requested message arrives. This is probably the most useful combination. It allows a process to send one or more messages to a variety of destinations as quickly as possible. A process that must receive a message before it can do useful work needs to be blocked until such a message arrives. An example is a server process that exists to provide a service or resource to other processes.
* Nonblocking send, nonblocking receive: Neither party is required to wait.

[14] Is busy waiting always less efficient (in terms of using processor time) than a blocking wait? Explain.

Answer:

On average, yes, because busy-waiting consumes useless instruction cycles.

However, in a particular case, if a process comes to a point in the program where it

must wait for a condition to be satisfied, and if that condition is already satisfied,

then the busy-wait will find that out immediately, whereas, the blocking wait will

consume OS resources switching out of and back into the process.

[15]

Answer:

The two are equivalent. In the definition of Figure 5.3, when the value of the

semaphore is negative, its value tells you how many processes are waiting. With

the definition of this problem, you don't have that information readily available.

However, the two versions function the same.

[16]

#define REINDEER 9 /\* max # \*/

#define ELVES 3 /\* size of /\* Semaphores \*/

only\_elves = 3, /\* 3 go to emutex = 1, /\* update rmutex = 1, /\* update rein\_semWait = 0, /\* block back from sleigh = 0, /\*all reindeer around done = 0, /\* toys all santa\_semSignal = 0, /\* 1st 2 elves on

this outside Santa's \*/

santa = 0, /\* Santa blocked \*/

problem = 0, /\* semWait the

question elf\_done = 0; /\* receive /\* Shared Integers \*/

rein\_ct = 0; /\* # of reindeer \*/

elf\_ct = 0; /\* # of elves with \*/

/\* Reindeer Process for (;;) {

tan on the beaches in the Pacific Christmas is close

semWait (rmutex)

rein\_ct++

if (rein\_ct == REINDEER) {

semSignal (rmutex)

semSignal (santa)

}

else {

semSignal (rmutex)

semWait (rein\_semWait)

}

/\* all reindeer semWaiting to be attached sleigh \*/

semWait (sleigh)

fly off to deliver toys

semWait (done)

head back to the Pacific islands

} /\* end "forever" loop \*/

/\* Elf Process for (;;) {

semWait (only\_elves) "in" \*/

semWait (emutex)

elf\_ct++

if (elf\_ct == ELVES) semSignal (emutex)

semSignal (santa) Santa \*/

}

else {

semSignal (emutex)

semWait (santa \_semWait outside

}

semWait (problem)

ask question /\* Santa semWait (elf\_done)

semSignal (only\_elves)

} /\* end "forever" loop \*/

/\* Santa Process for (;;) {

semWait (santa) /\* mutual exclusion is not because if it is not equal then elves woke up Santa if (rein\_ct == REINDEER) semWait (rmutex)

rein\_ct = 0 /\* reset semSignal (rmutex)

for (i = 0; i < REINDEER semSignal (rein\_semWait)

for (i = 0; i < REINDEER; semSignal (sleigh)

deliver all the toys and for (i = 0; i < REINDEER; semSignal (done)

}

else { /\* 3 for (i = 0; i < ELVES – semSignal (santa\_semSignal)

semWait (emutex)

elf\_ct = 0

semSignal (emutex)

for (i = 0; i < ELVES; semSignal (problem)

Answer: that question

semSignal (elf\_done)

}

}

} /\* end "forever" loop \*/

By,

Shreeya Vachhani (121052)