CSE 380: Homework 2: Synchronization

Due: Thursday, October 2, 2003

Submit a hardcopy solution of the problems in class on Oct 2, and submit code and
documentation for the programs using the turnin command to the account cse380@eniac
by 11.59pm on Oct 2. The code should include comments which adequately describe the
solution, as described below.

Reminder: Copying the solution from a fellow student, a website, or some other source,
is a violation of the University policies on academic integrity.

Implementing Dining Philosophers

Recall the dining philosophers problem of philosophers seated around a table eating spaghetti
periodically. In this question, you have to implement a solution using system calls for
semaphores. You will submit two variations of the dining philosophers problem:

a) a solution to the dining philosophers problem using processes, shared memory, and
semaphores

b) a solution using threads and semaphores.

A skeleton for each part is available at:

http://www.cis.upenn.edu/~cse380/hw2.tar.gz

It specifies the basic routines that need to be implemented. The skeleton includes a makefile
that will allow you to type make proc to generate the part one and make thread to generate
the program for part two. Please do not change the name of the files or the targets in the
makefile.

This assignment uses POSIX semaphores, Solaris shared memory functions, and POSIX
threads. You are expected to use these calls for your programs.

As far as the solution to the problem is concerned, you can use class notes, textbook,
or design your own solution.

Part One: Using Process and Shared Memory

For part one, your solution should work in the following way (see dinPhil_proc.c in the
skeleton):

1. Create one process for each philosopher

2. The number of philosophers should be the first argument to your program. The input
range is restricted from three to 20 philosophers, inclusive.

3. Each philosopher should eat bites times, which is the second input parameter to the
program. Count the number of times each philosopher eats, and when (s)he has eaten
the maximum number of bites, (s)he stops eating (the corresponding process exits).
4. For mutual exclusion, use POSIX semaphores. Relevant system calls are: `sem_init()`, `sem_post()`, `sem_wait`, `sem_getvalue` and `sem_destroy()`.
   The semaphores should be allocated in shared memory (see Appendix A on the system support for doing this in Solaris).

5. Simulate thinking and eating using the `random` and `sleep` system calls.

6. Your code should be well documented with sufficient explanation of high-level objectives as well as of details. Your documentation will serve as the main justification for the correctness of your program. It is your responsibility to convince the graders that your solution makes sense.

Part Two: Using Threads

Threads should be created using POSIX threads (`man pthreads` on eniac. An example of a creating pthreads is shown in Appendix B. Part two has the same basic requirements for part one with the following exceptions:

1. Create one thread for each philosopher instead of a process for each.
2. Shared memory is not required.
3. Mutual exclusion for the pthreads version can be implemented using either semaphores as in part one or by using `pthread_mutex_lock` and `pthread_mutex_unlock`.

Submission

We have created specific requirements for the naming of your source files, your program, the output, and the compilation process. Please do not remove targets from the makefile, or alter the names of the source files. Your hardcopy should divided into two sections, one for each part of the assignment. Separately staple each section and include your name, email address, and a designation of the part of the assignment at the beginning of each of the two sections. Please include an explanation before your code to explain your solution.

- use turnin with the following syntax:

  `turnin -ccse380 -phw2 Makefile dinPhil_thread.c dinPhil_proc.c`

- For every critical section in your code, add statements that identify the entering and exiting from that section. If your program has two critical sections, then the lock of the mutex before the first should be directly followed by the line:

  `printf("Starting critical section 1\n");`

- Unlocking the mutex to exit the critical section should be directly preceded by this line:

  `printf("Stopping critical section 1\n");`
Use the exact format above.

- When philosopher ‘i’ takes their forks, print a line with the following format to stdout:

  ```c
  printf("Philosopher %d taking forks\n", i);
  ```

  Similarly, when philosopher ‘i’ puts their forks down, print a line with the following format:

  ```c
  printf("Philosopher %d putting forks\n", i);
  ```

- Keep track of each time a philosopher eats. When philosopher ‘i’ eats for the ‘j’-th time, print the following:

  ```c
  printf("Philosopher %d eating for time %d\n", i, j);
  ```

  Use the man pages (as well as class notes and textbook) for information on necessary system calls (e.g. `man semaphore` and `man pthreads`).
Appendix A: Shared Memory System Calls

This section gives information on how Unix processes can request and use shared memory segments. For every shared memory segment, the kernel maintains the following structure of information:

```c
/*
 * There is a shared mem id data structure (shmid_ds) for each
 * segment in the system.
 */
struct shmid_ds {
    struct ipc_perm shm_perm; /* operation permission struct */
    size_t shm_segsz; /* size of segment in bytes */
    /*.... some implementation dependent info */
    pid_t shm_lpid; /* pid of last shmop */
    pid_t shm_cpid; /* pid of creator */
    shmatt_t shm_nattch; /* current # attached */
    ulong_t shm_cnattch; /* in-core # attached */
    time_t shm_atime; /* last shmat time */
    time_t shm_dtime; /* last shmdt time */
    time_t shm_ctime; /* last change time */
};
```

The `ipc_perm` structure contains the access permissions for the shared memory segment.

```c
struct ipc_perm {
    uid_t uid; /* owner’s user id */
    gid_t gid; /* owner’s group id */
    uid_t cuid; /* creator’s user id */
    gid_t cgid; /* creator’s group id */
    mode_t mode; /* access modes */
    uint_t seq; /* slot usage sequence number */
    key_t key; /* key */
};
```

The `shmget` function is defined as:

```c
int shmget(key_t key, int size, int shmflag);
```

The `shmflag` argument specifies the low-order 9 bits of the `mode` for the shared memory, and whether a new segment is being created or if an existing one is being referenced.

The `shmflag` argument is a combination of the constants:
### Description of Permissions

<table>
<thead>
<tr>
<th>Numeric</th>
<th>Symbolic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0400</td>
<td>SHM_R</td>
<td>Read by owner</td>
</tr>
<tr>
<td>0200</td>
<td>SHM_W</td>
<td>Write by owner</td>
</tr>
<tr>
<td>0040</td>
<td>SHM_R&gt;&gt;3</td>
<td>Read by group</td>
</tr>
<tr>
<td>0020</td>
<td>SHM_W&gt;&gt;3</td>
<td>Write by group</td>
</tr>
<tr>
<td>0004</td>
<td>SHM_R&gt;&gt;6</td>
<td>Read by world</td>
</tr>
<tr>
<td>0002</td>
<td>SHM_W&gt;&gt;6</td>
<td>Write by world</td>
</tr>
<tr>
<td></td>
<td>IPC_CREAT</td>
<td>See below</td>
</tr>
<tr>
<td></td>
<td>IPC_EXCL</td>
<td>See below</td>
</tr>
</tbody>
</table>

The rules for whether a new shared memory segment is created or whether an existing one is referenced are:

- Specifying a *key* of IPC_PRIVATE guarantees that a unique IPC channel is created.
- Setting the IPC_CREATE bit of the *shmflag* word creates a new entry for the specified *key*, if it does not exist. If an existing entry is found, that entry is returned.
- Setting both the IPC_CREATE and IPC_EXCL bits of the flag word creates a new entry of the specified key, only if the entry does not already exist. If an existing entry is found, an error occurs, since the IPC channel already exists.

Upon creation, the data structure associated with the new shared memory identifier is initialized as follows:

- The values of *shm_perm.cuid*, *shm_perm.uid*, *shm_perm.cgid*, and *shm_perm.gid* are set equal to the effective user ID and effective group ID, respectively, of the calling process.
- The access permission bits of *shm_perm.mode* are set equal to the access permission bits of *shmflag*. *shm_segsz* is set equal to the value of *size*.
- The values of *shm_lpid*, *shm_nattch* *shm_atime*, and *shm_dtime* are set equal to 0.
- The *shm_cftime* is set equal to the current time.

The following example illustrates how shared memory can be used.

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main (int argc, char *argv[]){
    int shmid; /* shared memory ID */
    char *p;  /* pointer to shared memory area */

    /* reserve a 10-byte physical memory segment using the pid as the key */
```
shmid = shmget ((key_t) getpid (), 10, 0666|IPC_CREAT);
if (shmid == -1) {
    puts ("shmget failed");
    exit (1);
}

/* attach the shared memory for use by this program */
p = (char *) shmat (shmid, (char *) 0, 0);
strcpy (p, "hello"); /* put string into shared memory */
puts (p);

/* detach the shared memory */
shmdt (p);

/* remove the shared memory */
shmctl (shmid, IPC_RMID, 0);
Appendix B: Pthreads Example: thread_test.c

This program creates two pthreads which share the variable counter. Threads are created using pthread_create which is passed, among other arguments, a pointer to a function that the thread should begin execution at as well as a pointer to an optional argument to that function. For a full explanation, see man pthread_create. In this case, the function that each thread will begin execution at is go() and the argument to the function is a pointer to an element in the array arg. Similar to wait() for process, join() will wait for the specified thread to exit. For a general overview of pthreads, see man pthreads and for specific functions, see the appropriate man page.

Semaphore can be used in pthreads and simple mutexes can be created as shown below. In this example, the variable counter must be protected by a mutex because two different threads are incrementing the value. Without the mutex, the potential for the counter to be incorrectly incremented exists. Although the probability of this occurrence is quite low in this example, as more threads are used and more data is shared, the probability will obviously increase.

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <pthread.h>
#include <errno.h>
#define NLOOP 5000
int counter;
pthread_mutex_t counter_mutex = PTHREAD_MUTEX_INITIALIZER;

void *go(void *);

main() {

    pthread_t tidA, tidB;
    int arg[2];
    arg[0] = 1; arg[1] = 2;

    pthread_create(&tidA, NULL, &go, (void *) &arg[0]);
    pthread_create(&tidB, NULL, &go, (void *) &arg[1]);

    /* wait for threads to stop */
    pthread_join(tidA, NULL);
```c
void
go(void *vptr) {
  int i, val;

  val = *(int *)vptr;
  printf("thread %d’s argument is %d\n", pthread_self(), val);

  pthread_mutex_lock(&counter_mutex);
  for (i = 0; i < NLOOP; i++) {
    counter++;
    printf("i = %d: id = %d : counter = %d\n", i, pthread_self(), counter);
  }
  pthread_mutex_unlock(&counter_mutex);
}
```