**Process Control Part 2 Inter-Process Communication**

|  |  |
| --- | --- |
| **Inter-Process Communication (IPC)**  It is very important for processes to be able to communicate with other processes. Mechanisms:  **Files** processes can read/write from files. Simultaneous usage involves direct access capabilities or databases.  **Pipes** One-way communication achieved by a process writing requests to the pipe and another process reading from the pipe. Requests queue up until the reader reads the pipe. The processes must be on the same computer.  **Named Pipes** Multiple processes can read and write from/to this file-like implementation of a pipe. The processes must be on the same computer.  **Shared Memory** Multiple processes can interact with a shared segment of memory. To avoid integrity issues, the shared memory must be protected. The processes must be on the same computer.  **Sockets** Data can be sent across a socket to another process which could be on the same computer or across a network on another computer. | **Pipes:**    **Named Pipes:**    **Shared Memory:**    **Sockets:** |
| **Pipes**  Previously, we showed how shells provide pipes to take the stdout of one process as the stdin of the next process. This is possible because Linux provides an inter-process communication mechanism called pipes. Linux pipes:   * One or more processes write to the pipe. * One process reads from the pipe. * If the writer(s) are writing data faster than the reader can consume it, Linux buffers the data. This allows the reader to catch up. * If the reader is reading before a writer writes data, the reader **blocks**. * A single read by a reader may actually read data from multiple writes or part of a write. A linefeed is not used as a limit. It is easier to manage read/write if the size of what is written is the same as the size of what is read.   This is probably the easiest IPC mechanism to implement, but is confusing if the read and write sizes are different.  To create a pipe:  int **pipe** (int fdM[]) - creates a pipe, returns the reader's file descriptor in fdM[0], returns the writer's file descriptor in fdM[1], and functionally returns -1 if it failed.  fdM[] is the array for file descriptor indexes from the file descriptor table for read and write  Examine the program on the right. The program must create the **pipe before it forks** the child. Since the size of the records being written is the same as the number of bytes read, the reader works properly. The writer (the parent) individually writes the command arguments to the pipe. The reader (the child) reads them and prints them  Compile it and name the executable ex21.  $ ./ex21 eleven twelve thirteen fourteen fifteen  **reader received: 'eleven'**  **reader received: 'twelve'**  **reader received: 'thirteen'**  **reader received: 'fourteen'**  **reader received: 'fifteen'**  **What happens if the writer writes a different size from what the reader reads?**  Replace the write REC\_SZ with iWriteSz, compile it and run it again,  $ ./ex21 eleven twelve thirteen fourteen fifteen  **reader received: 'eleventwelvethirteen'**  **reader received: 'fourteenfifteen'**  why is the output different?? Because the write only write the the size of the write not the REC\_SIZE = 20 🡺 so the read have to wait for the buffer to be full till 20 before the read  This can only be terminate(EOF) when the process close the write file  Why the parent wait??  Because we dont want no orphan or zombie, good practice | Example 2-1: parent writes to the pipe and child reads from it  #include <stdio.h>  #include <unistd.h>  #include <errno.h>  #include <string.h>  #define REC\_SZ 20  int main (int argc, char \*argv[])  {  long lForkPid;  int iExitStatus = 0; // Used by parent to get status of child  long lWaitPid; // PID of child that terminated  int iReadFd, iWriteFd; // File Descriptors for reader and writer  int iReadSz; // -1 (error) or size returned by read  int iWriteSz; // size to write  int fdM[2]; // contains file descriptors for the pipe  char szInput[REC\_SZ+1]; // buffer for reader  char szOut[REC\_SZ]; // buffer for writer  int i;  // create the pipe  if (pipe(fdM) == -1)  errExit("pipe not created: %s", strerror(errno));  iReadFd = fdM[0]; // reader file descriptor  iWriteFd = fdM[1]; // writer file descriptor  lForkPid = fork(); // create a child process  // Both the parent and child continue here  switch(lForkPid)  {  case -1:  errExit("fork failed: %s", strerror(errno));  case 0: // child process - the reader  close(iWriteFd); // reader doesn't need writer FD  iReadSz = read(iReadFd, szInput, REC\_SZ);  while (iReadSz > 0)  {  szInput[iReadSz] = '\0';  printf("reader received: '%s'\n", szInput);  iReadSz = read(iReadFd, szInput, REC\_SZ);  }  if (iReadSz == -1)  errExit("read error: %s", strerror(errno));  close(iReadFd);  break;  default: // parent process - the writer  close(iReadFd); // writer doesn't need reader FD  for (i=1; i < argc; i++)  {  memset(szOut, 0, REC\_SZ);  iWriteSz = strlen(argv[i])> REC\_SZ ? REC\_SZ : strlen(argv[i]);  memcpy(szOut, argv[i], iWriteSz);  write(iWriteFd, szOut, REC\_SZ);  }  close (iWriteFd);  lWaitPid = wait (&iExitStatus);  }  return 0;  } |
| **Pipelining in C**  When we discussed shell, we saw examples like this  ls /usr/bin | more  bash is, of course, just a program. How does bash invoke pipes to accomplish the pipelining?   * invokes **pipe** to create a pipe * **fork**s a child for each command * within each process, invokes **dup2** (see below) to redirect stdin/stdout to the pipe * within each process, invokes an **exec** function to execute the command   To redirect a file descriptor to another file descriptor, we will use dup2:  int **dup2** (int iFdFrom, int iFdTo)  redirects file descriptor for iFdTo to point to the same file descriptor as iFdFrom. If the file descriptor entry for iFdTo was already open, it closes that entry and then copies the entry of iFdFrom to iFdTo. dup2 returns -1 if it failed.  The program on the right implements a pipe for  "ls -l /usr/bin | sort -k5 -n":   * creates a pipe for the children to use * forks two children. * first child is used for "ls -l /usr/bin":   + invokes dup2 to redirect its stdout to the pipe   + invokes execl for that command * second child is used for "sort -k5 -n":   + invokes dup2 to redirect its stdin from the pipe   + invokes execl for that command * parent closes the pipe since it isn't using it. This has no negative impact on the children; however, leaving them open may cause the pipe reader to block waiting for any processes with an open write FD to complete. * parent also is writing to stderr to avoid interfering with the stdout produced by sort.   $ ./ex22 | more  PID for ls: 32421, PID for sort: 32422  … shows files from ls sorted by size | **Example 2-2: pipelining: ls -l /usr/bin | sort -k5 -n**  #include <stdio.h>  #include <unistd.h>  #include <errno.h>  #include <string.h>  int main () {  long lForkPidLs, lForkPidSort;  int iExitStatus = 0; // Used by parent to get status of child  long lWaitPid; // PID of child that terminated  int iReadFd, iWriteFd; // File Descriptors for reader and writer  int fdM[2]; // contains file descriptors for the pipe  // create the pipe  if (pipe(fdM) == -1)  errExit("pipe not created: %s", strerror(errno));  lForkPidLs = fork(); // create a child process  // Both the parent and first child continue here  switch(lForkPidLs) {  case -1:  errExit("fork failed: %s", strerror(errno));  case 0: // child process - ls -l /usr/bin/  if ( dup2(fdM[1], STDOUT\_FILENO) == -1)  errExit("Failed to rediect stdout for 'ls': %s"  , strerror(errno));  close(fdM[0]); close(fdM[1]);  execl("/bin/ls", "ls", "-l", "/usr/bin", NULL);  errExit("Failed to exec 'ls -l /usr/bin': %s"  , strerror(errno));  default: // parent process  // create the other child  lForkPidSort = fork(); // create a child process  // Both the parent and second child continue here  switch(lForkPidSort) {  case -1:  errExit("fork of second child failed: %s"  , strerror(errno));  case 0: // child process - sort -k5 -n  if ( dup2(fdM[0], STDIN\_FILENO) == -1)  errExit("Failed to rediect stdin for 'sort': %s"  **, strerror(errno));**  **close(fdM[0]); close(fdM[1]);**  **execl("/usr/bin/sort", "sort", "-k5", "-n", NULL);**  **errExit("Failed to exec 'sort -k5 -n': %s"**  **, strerror(errno));**  **default: // parent process**  **close(fdM[0]); close(fdM[1]);**  **fprintf(stderr, "PID for ls: %ld, PID for sort: %ld\n"**  **, lForkPidLs, lForkPidSort);**  **fflush(stderr);**  **}**  **}**  **return 0;**  **}** |
| **Waiting for multiple children**  The code to the right replaces the bottom switch default clause of example22. It waits for two children to complete.  $ ./ex23 | more  PID for ls: 32439, PID for sort: 32440  First child terminated  … shows files from ls sorted by size  Second child terminated  complete | **Example 2-3: pipelining: waiting for the children to complete**  **default: // parent process**  **close(fdM[0]); close(fdM[1]);**  **fprintf(stderr, "PID for ls: %ld, PID for sort: %ld\n"**  **, lForkPidLs, lForkPidSort);**  **fflush(stderr);**  **int i;**  **for (i = 0; i < 2; i+=1) {**  **iExitStatus = 0;**  **lWaitPid = wait(&iExitStatus);**  **if (lWaitPid == -1)**  **errExit("wait failed: %s\n"**  **, strerror(errno));**  **if (lWaitPid == lForkPidLs)**  **fprintf(stderr,"First child terminated\n");**  **else**  **fprintf(stderr,"Second child terminated\n");**  **fflush(stderr);**  **}**  **fprintf(stderr,"complete\n");** |
| **Named Pipes (FIFOs)**  Unlike pipes, named pipes can persist after the readers and writers terminate. Named pipes are files in the file system. They are also called FIFOs since the data is read First-In First-Out.  A server process usually creates the named pipe (using mkfifo) and acts as the reader.  The code for the server is on the right.  . | **Example 2-4: Server creates the named pipe (FIFO)**  $ vi pipeServer.c  #include <errno.h>  #include <fcntl.h>  #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <sys/stat.h>  #include <string.h>  #define FIFOARG 1  #define FIFO\_PERMS (S\_IRWXU | S\_IWGRP| S\_IWOTH)  int main (int argc, char \*argv[]) {  int iFifoFd;  if (argc != 2) //pipeS namedPipe  errExit("Usage: %s namedPipe > logfile\n", argv[0]);  // create a named pipe to handle incoming requests  if ((mkfifo(argv[FIFOARG], FIFO\_PERMS) == -1) && (errno != EEXIST))  errExit("Server failed to create a FIFO named '%s'", argv[FIFOARG]);  // open the FIFO as read/write so that pipe server won't  // terminate if there isn't anything to read  if ((iFifoFd = open(argv[FIFOARG], O\_RDWR)) == -1)  errExit("Server failed to open FIFO '%s'", argv[FIFOARG]);  // Handle all the requests  handleRequests(iFifoFd, STDOUT\_FILENO);  return 1; // should not reach here, so return failure  } |
| The code for **handleRequests.c** is on the right. It reads from a file specified by iFromFd file descriptor and writes to the file specified by the iToFd. | **Example 2-4 continued**  $ vi handleRequests.c  #include <unistd.h>  #define BLKSIZE 1024  void handleRequests(int iFromFd, int iToFd)  {  char szBuf[BLKSIZE]; // general buffer for the requests  int iBytesRead, iBytesWritten;  while (1)  {  // Read the request from the client  if ((iBytesRead = read(iFromFd, szBuf, BLKSIZE)) <= 0)  break;  // write it to a log  if ((iBytesWritten = write(iToFd, szBuf, iBytesRead)) == -1)  break;  }  }  $ gcc -o fifoS pipeServer.c handleRequests.c errExit.o |
| **Named Pipes (FIFOs) continued**  A client makes requests of the server by writing to the named pipe.  In the example code on the right, we pass the client the name of a named pipe and a client name (e.g., "A"). It opens the FIFO for read. If the server isn't currently reading, the client blocks on the open call.  To show the server and clients working, launch multiple terminal windows.  In one terminal window, launch the server and tell it to create a pipe named "RPipe":  $ ./fifoS RPipe  It will block while waiting for a client's message.  In another terminal window, launch a client:  $ ./fifoC RPipe A  It will write a message stating "A", its PID, and the current time. The server will show that message.  In another terminal window, launch another client:  $ ./fifoC RPipe B  It will write a message stating "B", its PID, and the current time. The server will show that message.  You can have each of the client terminal windows write several messages.  To terminate the server in the server's terminal window, press CTRL-C. | **Example 2-4 continued**  $ vi pipeClient.c  #include <errno.h>  #include <fcntl.h>  #include <limits.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <time.h>  #include <unistd.h>  #include <sys/stat.h>  #define BLKSIZE 1024  #define FIFOARG 1  int main (int argc, char \*argv[]) {  time\_t timeCurrent;  int iLen;  char szBuf[BLKSIZE+1]; // general buffer for the requests  int iFifoFd;  if (argc != 3) // pipeC pipeNm  errExit( "Usage: %s fifoName clientName\n", argv[0]);  // open the named pipe  if ((iFifoFd = open(argv[FIFOARG], O\_WRONLY)) == -1)  errExit("Client could not open FIFO '%s'", argv[FIFOARG]);  timeCurrent = time(NULL);  snprintf(szBuf, PIPE\_BUF, "%s %d: %s"  , argv[2]  , (int)getpid()  , ctime(&timeCurrent));  iLen = strlen(szBuf);  if (write(iFifoFd, szBuf, iLen) != iLen)  errExit("Client write error to FIFO '%s'", argv[FIFOARG]);  close(iFifoFd);  return 0;  }  $ gcc -o fifoC pipeClient.c errExit.o |
| **Request Response Using Named Pipes**  If we need the server to respond to a client request, we could create a named pipe for sending the response. The client could then read the response.  The code on the right shows a different version of handleRequests, but the code is in handleReqResp.c. The code for pipeServer.c didn't change.  handleReqResp.c:   * reads the request from the client and writes it to a log. * creates a name for the response pipe using the client name (which is part of the request) * creates the new response FIFO for the client response using mkfifo * opens the new response FIFO * writes the response message to the new response FIFO * closes the new response FIFO. This is necessary to reuse the FD. This also allows supporting many clients. | **Example 2-5 Request Response**  $ vi handleReqResp.c  #include <errno.h>  #include <fcntl.h>  #include <unistd.h>  #include <stdio.h>  #include <string.h>  #include <sys/stat.h>  #define BLKSIZE 1024  #define MAX\_NAME 20  #define FIFO\_PERMS (S\_IRWXU | S\_IRGRP| S\_IROTH)  void handleRequests(int iFromFd, int iToFd) {  char szBuf[BLKSIZE+1]; // general buffer for the requests  int iBytesRead, iBytesWritten;  char szClientNm[MAX\_NAME+1]; // Name passed from client  char szClientFIFO[MAX\_NAME+1]; // Name of response FIFO  int iFifoRespFd; // Response FIFO File Descriptor idx  int iLen;  int iScanfCnt; // used by sscanf  while (1) {  // Read the request from the client  if ((iBytesRead = read(iFromFd, szBuf, BLKSIZE)) <= 0)  break;  // write it to a log  if ((iBytesWritten = write(iToFd, szBuf, iBytesRead)) == -1)  break;  **// create the name of the response FIFO**  iScanfCnt = sscanf(szBuf, "%20s", szClientNm);  if (iScanfCnt != 1)  errExit("Invalid client name: '%s'", szBuf);  snprintf(szClientFIFO, MAX\_NAME, "%sPipe", szClientNm);  // create the response FIFO  if ((mkfifo(szClientFIFO, FIFO\_PERMS) == -1) && (errno != EEXIST))  errExit("Server failed to create a response FIFO named '%s'"  , szClientFIFO);  // open the response FIFO as read/write  if ((iFifoRespFd = open(szClientFIFO, O\_RDWR)) == -1)  errExit("Server failed to open response FIFO '%s'"  , szClientFIFO);  // Build the response and write it to the FIFO  snprintf(szBuf, BLKSIZE, "Response to %s, message read", szClientNm);  iLen = strlen(szBuf)+1;  if (write(iFifoRespFd, szBuf, iLen) != iLen)  errExit("Client write error to FIFO '%s'", szClientFIFO);  close(iFifoRespFd);  }  }  $ gcc -o pipeSR pipeServer.c handleReqResp.c errExit.o |
| **Request Response Using Named Pipes continued**  The client can read the response from the server by reading from a different named pipe.  To show the execution, we can proceed as was done in example 2-2, but we need to use fifoSR for the server, fifoCR for the client, and RRpipe for the request response pipe. | **Example 2-5 continued**  $ vi pipeClientResp.c  # #include <errno.h>  #include <fcntl.h>  #include <limits.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <time.h>  #include <unistd.h>  #include <sys/stat.h>  #define FIFOARG 1  #define MAX\_NAME 20  #define BLKSIZE 1024  int main (int argc, char \*argv[]) {  time\_t timeCurrent;  int iLen;  char szBuf[BLKSIZE+1]; // general buffer for the requests  int iFifoFd, iFifoRespFd;  char szRespFifoNm[MAX\_NAME+1]; // Response FIFO name  int iBytesRead;  if (argc != 3) // pipeC pipeNm  errExit( "Usage: %s fifoName clientName\n", argv[0]);  // open the named pipe  if ((iFifoFd = open(argv[FIFOARG], O\_WRONLY)) == -1)  errExit("Client could not open FIFO '%s'", argv[FIFOARG]);  timeCurrent = time(NULL);  snprintf(szBuf, PIPE\_BUF, "%s %d: %s"  , argv[2]  , (int)getpid()  , ctime(&timeCurrent));  iLen = strlen(szBuf);  // write a request to the server  if (write(iFifoFd, szBuf, iLen) != iLen)  errExit("Client write error to FIFO '%s'", argv[FIFOARG]);  **// create the response FIFO name and open it**  snprintf(szRespFifoNm, MAX\_NAME, "%sPipe", argv[2]);  if ((iFifoRespFd = open(szRespFifoNm, O\_RDWR)) == -1)  errExit("Client could not open response FIFO '%s'", szRespFifoNm);  // read the response from the server  if ((iBytesRead = read(iFifoRespFd, szBuf, BLKSIZE)) > 0)  printf("client received message: %s\n", szBuf);  close(iFifoFd);  close(iFifoRespFd);  return 0;  }  $ gcc -o pipeCR pipeClientResp.c errExit.o |
| **Shared Memory**  Processes can share information using a shared memory segment. This is the most efficient mechanism for processes to share data. Multiple processes can read and write the shared memory; however, shared memory doesn't queue up requests to write to it. If there are multiple processes writing to it, a separate mechanism must be used to protect it.  In our discussion, we will assume that the processes are independent (i.e., one isn't an ancestor of the other). It is slightly easier to share memory between related processes.  To use the same shared memory segment, it must be identified by a key (of type key\_t). | Shared Memory: |
| **Process for Using Shared Memory**  1. Obtain a common key. This can be done using **ftok**(*fileName*,1) to receive a key.  2. Use **shmget**(*key*, *size*, *flags*) to create or get the shared memory receiving a shmid.  3. Use **shmat**(*shmid*, NULL, 0) to attach the memory and receive shmptr (a pointer to the shared memory). This is similar to receiving a pointer from **malloc**.  4. Use the memory like you would with other pointers; however, changing the memory can be an issue since simultaneous changes from multiple processes can corrupt memory.  5. Use **shmdt**(*shmptr*) to detach the memory from your program's address space.  6. The last process to detach the shared memory segment should deallocate it by invoking **shmctl**(*shmid*, IPC\_RMID, NULL). |  |
| **SHM Steps 1 and 2 – Creating the Shared Memory**  For independent processes to use the same shared memory segment, it must be identified by a key\_t key using ftok:  key\_t **ftok** (const char \**pszFileNm*, int *iKey*)  The fully-qualified filename must exist for **ftok** to return a generated key. The combination of the file name and iKey (which is usually 1) is used to uniquely define the generated key. It returns -1 and sets errno on failure.  ftok did not create a shared memory segment; instead, it gave us a key to use to get it using **shmget**. To either create the shared memory or simply get access to it, we use **shmget**:  int **shmget**(key\_t *key*, int *size*, int *iFlags*)  The key is used to get the shared memory segment. If it doesn't exist and is to be created, *iFlags* should include IPC\_CREAT. **shmget** returns an int which is used to attach the shared memory. If it doesn't exist and isn't being created, it fails. shmget returns -1and sets errno on failure.   * The following constants can be bitwise or'd with the permission values:   + IPC\_CREAT - creates a file if it doesn't exist.   + IPC\_EXCL - gives an error if the file already exists * Permission flags can be or'd with those. Constants according to the POSIX standard:   + S\_IRUSR - read by owner   + S\_IWUSR - write by owner   + S\_IXUSR - execute by owner   + S\_IRGRP - read by group   + S\_IWGRP - write by group   + S\_IXGRP - execute by group   + S\_IROTH - read by others   + S\_IWOTH - write by others   + S\_IXOTH - execute by others | **Example 2-6: Using ftok to generate a shared memory key**  # This code assumes that the specified file exists.  #include <sys/ipc.h>  #include <sys/shm.h>  #include <sys/stat.h>  #include <string.h>  #include <errno.h>  ...  #define SHM\_NM "/tmp/abc123.sharedMemory"  ...  Data \*pData;  key\_t keySharedMem;  int ishmid;  // get a key for for independent processes to use  keySharedMem = ftok(SHM\_NM, 1);  if ( (key\_t) keySharedMem == -1)  errExit("Failed to derive key from '%s':%s", SHM\_NM, strerror(errno));  // create or get the shared memory using IPC\_CREAT.  // If your process should only get it, don't specify IPC\_CREAT.  // Note that Data is just a typedef for data that we want in  // the shared memory. It is something we have to define.  ishmid = shmget(keySharedMem, sizeof(Data), IPC\_CREAT  |S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IWGRP);  if ( ishmid == -1 )  errExit("Failed to get shared mem:", strerror(errno)); |
| **SHM Step 3 - Receiving a Pointer to Shared Memory**  To receive a pointer to the shared memory, **shmat** is used.  void \***shmat**(int *shmid*, NULL, 0)  Returns a pointer to the shared memory for the shmid which came from a **shmget** call. On failure, it returns (void \*) -1. | **Example 2-6 continued: Step 3 - attaching to the shared memory**  // get a pointer to our Data  pData = shmat(ishmid, NULL, 0);  // On failure we have to check if the pointer is -1. Since this isn't  // a numeric type, we have to compare it with a type cast of -1.  if (pData == (void \*)-1)  errExit("Failed to attach shared mem:", strerror(errno)); |
| **SHM Step 5 - Detaching and Freeing the Memory**  To detach the shared memory, **shmdt** is used  int **shmdt**(void \**shmptr*)  Dettachs the shared memory that was previously returned by a call to **shmat**. On failure, it returns -1 and sets errno.  One process also needs to deallocate the shared memory segment using **shmctl**.  int **shmctl**(int *shmid*, IPC\_RMID, NULL)  Deallocates the shared memory segment. On failure, it returns -1 and sets errno. | **Example 2-6 continued: Step 5 - Detaching and Freeing the Memory**  // detach the shared memory  if (shmdt(pData) == -1)  errExit("Detach Failed: %s", strerror(errno));  // deallocate the shared memory segment  if (shmctl(ishmid, IPC\_RMID, NULL) == -1)  errExit("Removal of shared memory failed: %s", strerror(errno)); |
| **Protecting Shared Memory and Race Conditions**  Depending on the changes, if two or more processes need to update shared memory, it is possible that the memory could get corrupted. A **race condition** is where two processes are each trying (i.e., racing) to get their changes and not be affected by the other process. To avoid that problem, changes have to be synchronized.  Examine example 2-7 for further understanding of a race condition.  Changes to resources must be atomic, not allowing others to touch the data until the changes are complete.  There are several approaches to synchronize including **locking**, **semaphores**, **mutex**, and **barriers**. This problem is further examined by CS3733 (OS) to protect memory/devices and CS3743 (DB Mgt) to protect data on disk storage. | **Example 2-7: race condition issue**  Suppose two processes are both updating a shared binary tree by making insertions. Without a mechanism to synchronize the changes, what could happen?     * Process A wants to add key 38 to the binary tree. It searches down to the node containing 40 as pointer pNode. It accesses pNode->pLeft which is NULL. * Process B wants to add key 35 to the binary tree. It searches down to the node containing 40 as pointer pNode. It accesses pNode->pLeft which is NULL. * Process A allocates a new node to contain 38. It sets pNode->pLeft to point to that new node containing 38. * Process B also allocates a new node to contain 35. It sets pNode->pLeft to point to its new node contianing 35.   What is the result?  Process A changes were lost |
| **Lock Variable Approach**  For simplicity (not necessarily an ideal solution), we will use a lock variable approach:   * One process is the server and one process is the client. The client waits until the server provides information. * Assume the shared memory has a variable used as a simple lock. (This isn't a database lock.) The server sets the lock variable to READY after it provides information. * The client sees the READY, consumes the data, and sets the lock variable to NOT\_READY. | |  |  |  | | --- | --- | --- | | **Step** | **Server** | **Client** | | 1 | Creates the shared memory:   * ftok * shmget …. IPC\_CREAT * shmat * Server sets the LockVar to NOT\_READY. | Acquires the shared memory:   * ftok * shmget … (not using IPC\_CREAT) * shmat | | 2 | Server places data in the shared memory. | Client waits for LockVar to be READY. | | 3 | Servers sets the LockVar to READY. | Client copies the data out of shared memory. | | 4 | Server waits for the LockVar to be NOT\_READY. | Client sets the Lockvar to NOT\_READY. | | 5 | Server process continues to step 2 until it is done. | Client continues with step 2. | |
| **Server Example**  We have a candy server that is providing candy to its client. Most of the code to set up the shared memory was shown in example 2.6.  To run this server:  1. Create the file /tmp/abc123.sharedMemory  2. To compile and link this code, use  make candyS  3. Run it using ./candyS  The server should show that it provided one type of candy.  What is the purpose of the call of msleep?   * ?? | **Example 2.8: candy server which uses shared memory to communicate with client**  The complete code is provided in candyServer.c, msleep.c, and errExit.c  // Mark the shared data as not ready for consumption  pData->cLockVar = NOT\_READY;  while (fgets(szInputBuffer, BUF\_SZ, pFile) != NULL)  {  // copy the data to the target variables  iScanfCount = sscanf(szInputBuffer, "%d %22[^\n]"  , &pData->iQty  , pData->szCandy);  if (iScanfCount < 2)  errExit("Only received %d valid values. Found : %s\n"  , iScanfCount, szInputBuffer);  // mark it as ready for consumption  printf("Server: made %d %s ready\n", pData->iQty, pData->szCandy);  pData->cLockVar = READY;  // wait for client to consume it and mark it NOT\_READY  while (pData->cLockVar == READY)  msleep(100); // sleep for 100 microsec  // See if client is full of candy  if (pData-> cLockVar == CLIENT\_FULL)  {  printf("Server: client reported that it is FULL\n");  break;  }  }  // detach the shared memory  if (shmdt(pData) == -1)  errExit("Detach Failed: %s", strerror(errno));  // deallocate the shared memory segment  if (shmctl(ishmid, IPC\_RMID, NULL) == -1)  errExit("Removal of shared memory failed: %s", strerror(errno));  return 0; |
| **Client Example**  We have a candy client which consumes that candy provided by the server. Most of the code to set up the shared memory was shown in example 2.6; however, the client didn't create the shared memory.  To run this client:  1. Create another terminal window.  2. To compile and link this code, use  **make candyC**  3. Show the two terminal windows side-by-side.  4. Run it using ./candyC  The client should show that it ate the candy. The server will then provide more candy until the client is full. | **Example 2.9: candy client which uses shared memory to communicate with server**  The complete code is provided in candyClient.c, msleep.c, and errExit.c  int iTotal = 0;  // Continue until client has gotten 100 pieces of candy.  while (TRUE)  {  // wait until it is server says it is ready  while (pData->cLockVar != READY)  msleep(100); // sleep for 100 microsec  // Consume the candy  printf("Client: ate %d %s ready\n", pData->iQty, pData->szCandy);  iTotal += pData->iQty;  if (iTotal > 100)  break;  pData->cLockVar = NOT\_READY;  }  printf("Client: full\n");  pData->cLockVar = CLIENT\_FULL;  if (shmdt(pData) == -1)  errExit("Detach Failed: %s", strerror(errno)); |
| **In CS3733, you will discuss other (better) techniques for protecting the shared memory.**  This simple variable lock works *OK* for one server and one client. | **Observations:**  Even though we are using a **msleep** call to let the operating system take control, the wait loop is wasteful of CPU. We need a better mechanism which waits and doesn't consume CPU unnecessarily.  What happens if a client doesn't consume the candy?   * ??   What would happen if we had multiple clients wanting the candy?   * ?? |
| **Cleaning Up Shared Memory**  System administrator should periodically look for shared memory which wasn't deallocated using the **ipcs** command.  It can then be removed using the **ipcrm** command. |  |

**©2017 Larry W. Clark,** UTSA CS students may make copies for their personal use