**/\*aio-nonsynchronized.c& aio-synchronized.c\*/ \* Program to Illustrate Async I/O with non-synchromized & synchromized writes \*/ /\* argv[1] file to write \*/ /\* artv[2] number of writes \*/ #define NUMBYTES 1024\*10240 #define PAUSE 1024\*1024 char buff[NUMBYTES]; void main (int argc, char \*argv[]) { int fd; struct aiocb cb; int count,numwrites,i,j; count=0; numwrites=atoi(argv[2]); /\*Use Non-Synchronized I/O \*/ if((fd=open(argv[1],O\_WRONLY|O\_CREAT|O\_TRUNC,0640)) == -1) {perror("Open failed: ");exit (1);} /\*Use Synchronized I/O \*/ if((fd=open(argv[1],O\_WRONLY|O\_CREAT|O\_TRUNC|O\_SYNC,0640)) == -1) { perror("Open failed: ");exit (1); }/\* Fill in AIO Control Block \*/ cb.aio\_fildes = fd; cb.aio\_buf = buff; cb.aio\_nbytes = NUMBYTES; cb.aio\_sigevent.sigev\_notify = SIGEV\_NONE; /\* No signals \*/ cb.aio\_offset = 0; cb.aio\_reqprio = 0; for (i=0; i<numwrites;i++){ aio\_write(&cb); while (aio\_error(&cb)) {/\* Waste Time - other work goes here \*/j=PAUSE; while (j) j--; count++;} lseek(fd,0,SEEK\_SET);} printf ("count=%d\n",count);}/\*chdate.c \*/ struct stat statbuf; struct utimbuf newt; stat(argv[1],&statbuf); printf("Lastaccessed: %s\n",ctime(&statbuf.st\_atime)); printf("Lastmodified: %s\n",ctime(&statbuf.st\_mtim)); printf("Last changed: %s\n",ctime(&statbuf.st\_ctime)); newt.actime = statbuf.st\_atime -600; newt.modtime= statbuf.st\_mtime +600; utime(argv[1],&newt); /\*copy.c\*/ int fd1,fd2; /\* file descriptors \*/ char buff [4096]; /\* overkill on buffer size\*/ int size; /\* real buffer size \*/ int iosize; /\* actual amount read \*/ if ((fd1=open(argv[1],0)) == -1) {perror("open1");exit(1);} if ((fd2=open(argv[2],O\_WRONLY | O\_CREAT |O\_TRUNC,0644)) == -1) {perror("open2");exit(1);} size=atoi(argv[3]); while ((iosize=read(fd1,buff,size)) >0 ) write(fd2,buff,iosize); if (iosize == -1) {perror("read"); exit(1);} /\*creat.c\*/ int fd;if ((fd=creat("abc",0))== -1) { perror("creat"); exit (1); }close(fd); /\*dirs.c\*/ DIR \*dirp; struct dirent \*dp; struct stat buf; if (!(dirp = opendir(argv[1]))) { /\* open the directory \*/ perror("opendir:"); exit(1); } chdir(argv[1]); /\* change to target directory, so filenames are relative to where we are \*/ while ((dp = readdir(dirp)) != NULL) {/\* print names/inodes \*/ printf("%-10d%s",(int)dp->d\_ino,dp->d\_name); if (stat(dp->d\_name,&buf) == -1) { /\* get inode info\*/ printf("\n"); perror("Bad Stat"); } /\* print modes \*/ printf("\tmode=%o R=%d D=%d B=%d C=%d L=%d\n", (int)buf.st\_mode, S\_ISREG(buf.st\_mode), S\_ISDIR(buf.st\_mode), S\_ISBLK(buf.st\_mode), S\_ISCHR(buf.st\_mode), S\_ISLNK(buf.st\_mode) ); } closedir(dirp); /\*stcopy.c\*/ /\* Program to copy a file, using stdio \*/ int main (int argc, char \*\*argv) { FILE \*fd1,\*fd2; /\* FILE descriptors \*/ int c; /\* input buffer \*/ if ((fd1=fopen(argv[1],"r")) == NULL) { puts("open1"); exit(1); } if ((fd2=fopen(argv[2],"w"))== NULL) { puts("open2"); exit(1); } while ((c=getc(fd1)) != EOF ) putc(c,fd2); fclose(fd2); } /\*prenv.c\*/ \* program to print the environment and pid \*/ int main(int argc, char \*argv[], char \*envp[]) { int i; printf("My PID is %d\nHere is my environment:\n",getpid()); for (i=0; envp[i] != NULL; i++) printf("%s \n",envp[i]); exit(0); } /\*spawn.c\*/ /\* program to illustrate process creation via fork and exec \*/ char parent[15]; /\* room for parents pid in ascii \*/ char \*child[]={"myname=john","myname=mary"}; /\* child's name \*/ int main(int argc, char \*argv[], char \*envp[]) { int i; /\* generic counter \*/ int pid[2]; /\*pids of children \*/ int whom; /\* pid of dead child\*/ int esize; /\* number of env variables \*/ char \*\*newenvp; /\* constructed env pointer \*/ int status; /\* chilsds return status\*/ /\* for fun we will build a new environment with some extra values \*/ sprintf( parent,"parent=%d", getpid()); /\* how big is the old environment \*/ for (esize=0; envp[esize] != NULL; esize++); /\* make new envp 2 slots longer \*/ newenvp=( char \*\*) malloc((esize+3)\*sizeof(char \*)); newenvp[0]=parent; /\* newenvp[1] will be set differently for each child \*/ for (i=0; i<=esize; i++) newenvp[i+2]=envp[i]; for (i=0; i<2; i++) { if ((pid[i]=fork()) == -1) { perror("fork"); exit (i+3); } if (pid[i] == 0) { /\* this is the child \*/ newenvp[1]=child[i]; /\* tell him his name \*/ sleep(getpid()%4); /\* random wait for some drama\*/ /\* we will redirect stdout \*/ close(1); if (open((i ? "mary" : "john"),O\_WRONLY |O\_CREAT |O\_TRUNC,0644) == -1) { perror("open stdout"); exit (2\*(1+i)); } /\* now exec the new program \*/ execle("prenv","prenv",NULL,newenvp); perror("execle"); /\* shoudn't get here \*/ exit(4\*(i+1)); } } /\* now parent waits \*/ for (i=0; i<=1; i++ ) { if ((whom=wait(&status)) == -1) { perror("wait"); exit(1); } if (whom == pid[0] ) printf("John died with status=%x\n",status); else if (whom == pid[1] ) printf("Alas poor Mary died with status=%x\n",status); else printf("Beats me who died, his pid is=%d",whom); } } /\*namesort.c\*/ int main(int argc, const char \*argv[]) { int fd[2]; pipe(fd); int sort\_pid=fork(); if(sort\_pid==0){ close(fd[1]); dup2(fd[0],0); close(fd[0]); int write\_file = open(argv[2],O\_CREAT | O\_WRONLY, 0666); dup2(write\_file, 1); execl("/usr/bin/sort", "sort", NULL); close(write\_file); } close(fd[0]); FILE \* read\_file = fopen(argv[1], "r"); FILE \* write\_to\_pipe = fdopen(fd[1], "w"); char read\_buf[1000]; char write\_buf[1000]; while (fgets(read\_buf, 1000, read\_file) != NULL) { int i; for (i=0; i < 1000; i++) { if (read\_buf[i] == ' ') break; } int j=0; i++; while (read\_buf[i]!='\n'){ write\_buf[j++] = read\_buf[i++]; } write\_buf[j++] = '\n'; write\_buf[j] = '\0'; fputs(write\_buf, write\_to\_pipe); } fclose(write\_to\_pipe); fclose(read\_file); close(fd[1]); /\*perfectthread.c\*/int checkit(int current\_num){ int sum = 0; int i; for(i = 1; i < current\_num; i++) { if (current\_num % i != 0) continue; sum += i; } return current\_num == sum; } void \* run\_compute(void\* start){ int i; int \* begin =(int \*) start; for(i=begin;;i++){ if(checkit(i)){ printf("perfect number: %d", i); } } } int main(int argc, char \*argv[]){ int start; while (scanf("%d",&start)){ pthread\_t temptid; pthread\_create(&temptid, NULL, run\_compute, &start); } return 0;}/\*perfectfork.c\*/#define KEY (key\_t)12347 int main(int argc, char \*argv[]){ int start; int begin; int status; int semid; /\* semaphore id \*/ struct sembuf sb; /\* semaphore buffer \*/ int process\_num=20; int j; for (j = 0; j < process\_num; j++){ int pid\_t = fork(); if (pid\_t ==0){ int i; int sid; semid=semget(KEY,1 ,IPC\_CREAT |0660); sb.sem\_op =-1; /\* set up for a lock operation\*/ sb.sem\_num =0; sb.sem\_flg =0; semop(semid, &sb, 1); sid=shmget(KEY,100\*sizeof (int),IPC\_CREAT |0660); start=(int) shmat(sid,0,0); for(i=start;;i++){ if(checkit(i)){ printf("perfect number: %d", i); } } shmdt(start); shmctl(sid,IPC\_RMID,0); } } while (scanf("%d",&begin)){ int sid; sid=shmget(KEY,100\*sizeof (int),IPC\_CREAT |0660); start=(int) shmat(sid,0,0); start = begin; semid=semget(KEY,1 ,IPC\_CREAT |0660); sb.sem\_op = 1; /\* set up for a unlock operation\*/ sb.sem\_num = 0; sb.sem\_flg = 0; semop(semid, &sb, 1); } while(wait(NULL) != -1); return 0; }/\*sem\_thread.c\*/typedef struct { int value, wakeups; pthread\_mutex\_t \*mutex; pthread\_cond\_t \*cond; } Semaphore; Semaphore \*make\_semaphore (int value) { Semaphore \*semaphore = malloc (sizeof(Semaphore)); semaphore->value = value; semaphore->wakeups = 0; semaphore->mutex = malloc (sizeof(pthread\_mutex\_t)); pthread\_mutex\_init (semaphore->mutex, NULL); semaphore->cond = malloc (sizeof(pthread\_cond\_t)); pthread\_cond\_init (semaphore->cond, NULL); return semaphore; } void sem\_wait (Semaphore \*semaphore) { pthread\_mutex\_lock (semaphore->mutex); semaphore->value--; if (semaphore->value < 0) { do { pthread\_cond\_wait (semaphore->cond, semaphore->mutex); } while (semaphore->wakeups < 1); semaphore->wakeups--; } pthread\_mutex\_unlock(semaphore->mutex); } void sem\_signal (Semaphore \*semaphore) { pthread\_mutex\_lock (semaphore->mutex); semaphore->value++; if (semaphore->value <= 0) { semaphore->wakeups++; pthread\_cond\_signal (semaphore->cond); } pthread\_mutex\_unlock(semaphore->mutex); }/\*sem\_mes.c\*/struct mesg\_buffer { long type; int pid; }; int queue[CAPACITY]; unsigned int size = 0; unsigned int rear = CAPACITY - 1; unsigned int front = 0; int enqueue(int data) { if (size == CAPACITY) { return 0; } rear = (rear + 1) % CAPACITY; size++; queue[rear] = data; return 1; } int dequeue() { int data; if (size == 0) { return data; } data = queue[front]; front = (front + 1) % CAPACITY; size--; return data; } int value=-1; const int processnum=20; int pids[processnum]; void sem\_signal(int pid) { int qid=msgget(KEY,IPC\_CREAT |0660); struct mesg\_buffer message; message.type = 0; message.pid = pid; msgsnd(qid,&message,sizeof(message.pid),0); } void sem\_wait(int pid) { int qid=msgget(KEY,IPC\_CREAT |0660); struct mesg\_buffer message; message.type = 1; message.pid = pid; msgsnd(qid,&message,sizeof(message.pid),0); msgrcv(qid,&message,sizeof(message.pid),0,0); } void synch(){ while (1) { int qid=msgget(KEY,IPC\_CREAT |0660); struct mesg\_buffer message; msgrcv(qid,&message,sizeof(message.pid),0,0); if (message.type == 0) value = value + 1; if (value <= 0) { int pid = dequeue(); message.type = 0; message.pid = pid; msgsnd(qid,&message,sizeof(message.pid),0); } else if (message.type == 1) { value = value - 1; if (value < 0) enqueue(message.pid); else msgsnd(qid,&message,sizeof(message.pid),0); } } }/\*signals.c\*//\* Program to illustrate the use of POSIX signals on UNIX The program runs a computational loop to compute perfect numbers starting at a fixed point. A time alarm signal is used to periodically print status An interrupt signal is used for status on demand A quit signal is used to reset the test interval (or terminate) void perfect(int); sigjmp\_buf jmpenv; /\* environment saved by setjmp\*/ int n; /\* global variable indicating current test point \*/ int main() { int begin; /\* starting point for next search\*/ /\* interrupt routines\*/ void status(); void query(); sigset\_t mask; struct sigaction action; if (sigsetjmp(jmpenv,0)) { printf("Enter search starting point (0 to terminate): "); scanf("%d",&begin); if (begin==0) exit(0); sigprocmask(SIG\_UNBLOCK, &mask, NULL); } else begin=2; /\* Status Routine will handle timer and INTR \*/ sigemptyset(&mask); sigaddset(&mask, SIGINT); sigaddset(&mask, SIGALRM); sigaddset(&mask, SIGQUIT); action.sa\_flags=0; action.sa\_mask=mask; action.sa\_handler=status; sigaction(SIGINT,&action,NULL); sigaction(SIGALRM,&action,NULL); action.sa\_handler=query; sigaction(SIGQUIT,&action,NULL); /\* start alarm clock \*/ alarm(20); perfect(begin); } void perfect(start) int start; { int i,sum; n=start; while (1) { sum=1; for (i=2;i<n;i++) if (!(n%i)) sum+=i; if (sum==n) printf("%d is perfect\n",n); n++; } } void status(signum) int signum; { alarm(0); /\* shutoff alarm \*/ if (signum == SIGINT) printf("Interrupt "); if (signum == SIGALRM) printf("Timer "); printf("processing %d\n",n); alarm(20); /\*restart alarm\*/ } void query() {siglongjmp(jmpenv,1);}/\*threadwith&withoutscop.c\*/void \*producer(),\*consumer(); int \*array; int main(int argc, char \*\*argv) { int howmany; pthread\_t johntid,marytid; pthread\_attr\_t tattr; long long \*howlong; howmany=atoi(argv[1]); array=calloc(howmany,sizeof(int)); pthread\_attr\_init(&tattr); pthread\_attr\_setscope(&tattr,PTHREAD\_SCOPE\_SYSTEM); pthread\_create(&marytid,&tattr, consumer, &howmany); pthread\_create(&johntid,&tattr, producer, &howmany); pthread\_join(johntid,NULL); pthread\_join(marytid,(void \*\*)&howlong); printf("John and Mary Threads done with wait %lld\n",\*howlong); } void \* producer(int \*howmany) { int i; sleep(1); for (i=0;i<\*howmany;i++) array[i]=i+1; printf("John produced %d Numbers\n",\*howmany); pthread\_exit(NULL); } void \*consumer(int \*howmany) { int i; long long sum=0; long long \*wait; wait =malloc (sizeof (long long)); \*wait=0; for (i=0;i<\*howmany;i++) {while (!array[i]) (\*wait)++; sum+=array[i];} printf("Mary consumed %d Numbers for a total of %lld\n",\*howmany,sum); pthread\_exit(wait); }/threadwithoutcopemain.c\*/int main(int argc, char \*\*argv) { int howmany; pthread\_t johntid,marytid; long long \*howlong; howmany=atoi(argv[1]); array=calloc(howmany,sizeof(int)); pthread\_create(&marytid,NULL, consumer, &howmany); pthread\_create(&johntid,NULL, producer, &howmany); pthread\_join(johntid,NULL); pthread\_join(marytid,(void \*\*)&howlong); printf("John and Mary Threads done with wait %lld\n",\*howlong); }/\*conditionvariable.c\*/int limit=-1; pthread\_mutex\_t mtx = PTHREAD\_MUTEX\_INITIALIZER; pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER; void \* producer(int \*howmany) { int i; for (i=0;i<\*howmany;i++) { array[i]=i; if (!i%1000) { pthread\_mutex\_lock(&mtx); limit=i; pthread\_cond\_signal(&cond); pthread\_mutex\_unlock(&mtx); } } pthread\_mutex\_lock(&mtx); limit=i-1; pthread\_cond\_signal(&cond); pthread\_mutex\_unlock(&mtx); printf("John produced %d Numbers\n",\*howmany); pthread\_exit(NULL); } void \*consumer(int \*howmany) { int i,mylimit; long long sum=0; long long \*wait; wait =malloc (sizeof (long long)); \*wait=0; i=0; while (i< \*howmany) { pthread\_mutex\_lock(&mtx); while (mylimit == limit) { (\*wait)++; pthread\_cond\_wait(&cond,&mtx); } mylimit=limit; pthread\_mutex\_unlock(&mtx); while (i<=mylimit) sum+=array[i++]; } printf("Mary consumed %d Numbers for a total of %lld\n",\*howmany,sum); pthread\_exit(wait); } /\*mutex.c\*/pthread\_mutex\_t mtx = PTHREAD\_MUTEX\_INITIALIZER; void \* producer(int \*howmany) { int i; for (i=0;i<\*howmany;i++) { array[i]=i; if (!i%1000) { pthread\_mutex\_lock(&mtx); limit=i; pthread\_mutex\_unlock(&mtx); } } pthread\_mutex\_lock(&mtx); limit=i-1; pthread\_mutex\_unlock(&mtx); printf("John produced %d Numbers\n",\*howmany); pthread\_exit(NULL); } void \*consumer(int \*howmany) { int i,mylimit; long long sum=0; long long \*wait; wait =malloc (sizeof (long long)); \*wait=0; i=0; while (i< \*howmany) { pthread\_mutex\_lock(&mtx); if (mylimit == limit) (\*wait)++; mylimit=limit; pthread\_mutex\_unlock(&mtx); while (i<=mylimit) sum+=array[i++]; } printf("Mary consumed %d Numbers for a total of %lld\n",\*howmany,sum); pthread\_exit(wait); }/\*johnfi.c\*//\* program to illustrate sending half of fifo on System V argument tells how many numbers to send to mary \*/ int main(int argc, char \*argv[]) { int fifofd; /\*file descriptor for fifo \*/ int j; /\*loop counter \*/ int n; /\* total number of items to send \*/ if (argc !=2) { printf("Usage: %s num\n",argv[0]); exit(3); } n=atoi(argv[1]); /\* make the fifo - ignore error if it exists \*/ if ((mkfifo("/tmp/john", 0666)== -1) && errno!= EEXIST) { perror("mkfifo"); exit(1); } /\* open fifo with O\_NONBLOCK so we won't block forever \*/ while ((fifofd=open("/tmp/john",O\_WRONLY |O\_NONBLOCK)) == -1) { if (errno == ENXIO) { printf("WHERE is that Mary\n"); sleep(20); } else { perror("fifo open"); exit(2); } } /\* John will generate numbers and Mary will add them\*/ for (j=1;j<=n;j++) while (write(fifofd, &j,sizeof (int))== -1) { printf("I seem to always wait for Mary\n"); sleep(20); } close(fifofd); /\* close the output fifo\*/ printf("I wrote %d number to Mary\n",n); }/\*maryfo.c \*//\* program to illustrate receiving half of fifo on System V \*/ int main() { int fifofd; /\*file descriptor for fifo \*/ int j; /\*Loop counter \*/ int sum; /\* sum of items sent \*/ int n; /\*input buffer \*/ j=1; /\* make the fifo - ignore error if it exists \*/ if ((mkfifo("/tmp/john",0666)== -1) && errno!= EEXIST) { perror("mkfifo"); exit(1); } /\* open fifo with O\_NONBLOCK so we won't block forever \*/ while ((fifofd=open("/tmp/john",O\_RDONLY |O\_NONBLOCK)) == -1) { perror("fifo open"); exit(2); } /\* John will generate numbers and Mary will add them\*/ /\* read one number without blocking \*/ while (read(fifofd,&sum,sizeof (int))==0) { printf("WHERE is that JOHN\n"); sleep(20); } fcntl(fifofd,F\_SETFL,O\_RDONLY); /\* cancel out O\_NOBLOCK \*/ /\*now read the rest \*/ while (read(fifofd, &n,sizeof (int))!= 0){ sum+=n; j++; } close(fifofd); /\* close the output fifo\*/ printf("I Mary read %d numbers totaling %d\n",j,sum); }}/\*johnsharesem.c\*//\* Program to illustrate sharing memory on System V \*/ /\* John will create a vector of numbers in shared memory \*/ /\* John will use a semaphore, \*/ /\* Also uses a semaphore to lock the region from Mary until John is done with it. Note this is only a one way lock.\*/ #define KEY (key\_t)12346 /\*key for shared memory segment \*/ int main() { int sid; /\* segment id of shared memory segment \*/ int \*array; /\* pointer to shared array, no storage yet\*/ int j; /\*loop counter \*/ int semid; /\* semaphore id \*/ struct sembuf sb; /\* semaphore buffer \*/ /\* create shared segment if necessary \*/ if ((sid=shmget(KEY,100\*sizeof (int),IPC\_CREAT |0660))== -1) { perror("shmget"); exit(1); } /\* map it into our address space\*/ if ((array=((int \*) shmat(sid,0,0)))== (int \*) -1) { perror("shmat"); exit(2); } /\* Now fill it up \*/ for (j=0;j<=100;j++) array[j]=j; /\* get semaphore id\*/ if ((semid=semget(KEY,1 ,IPC\_CREAT |0660))== -1) { perror("semget"); exit(1); } sb.sem\_op = 1; /\* set up for a unlock operation\*/ sb.sem\_num = 0; sb.sem\_flg = 0; if (semop(semid, &sb, 1) == -1) { /\* should not block \*/ perror("sem unlock"); exit(1); } }/\*marysharesem.c\*//\* Program to illustrate sharing memory on System V \*/ /\* Mary will read a vector of numbers in shared memory \*/ /\* Also uses a semaphore to lock the region from Mary until John is done with it. Note this is only a one way lock #define KEY (key\_t)12346 /\*key for shared memory segment \*/ struct sembuf sb; /\* semaphore buffer \*/ main() { int sid; /\* segment id of shared memory segment \*/ int \*array; /\* pointer to shared array, no storage yet\*/ int j; /\*loop counter \*/ int sum; /\*running sum\*/ int semid; /\* semaphore id \*/ /\* create shared segment if necessary \*/ if ((sid=shmget(KEY,100\*sizeof (int),IPC\_CREAT |0660))== -1) { perror("shmget"); exit(1); } /\* map it into our address space\*/ if ((array=((int \*) shmat(sid,0,0)))== (int \*)-1) { perror("shmat"); exit(2); } /\* get semaphore id\*/ if ((semid=semget(KEY,1 ,IPC\_CREAT |0660))== -1) { perror("semget"); exit(1); } sb.sem\_op =-1; /\* set up for a lock operation\*/ sb.sem\_num =0; sb.sem\_flg =0; if (semop(semid, &sb, 1) == -1) { /\* will block if locked \*/ perror("sem lock"); exit(1); } /\* Now add it up \*/ sum=0; for (j=0;j<=100;j++) sum+=array[j]; printf("Mary says arrary sun is %d\n",sum); /\* Unmap and deallocate the shared segment \*/ if (shmdt( (char \*) array) == -1) { perror("shmdt"); exit(3); } if (shmctl(sid,IPC\_RMID,0) == -1) { perror("shmctl"); exit(3); } } /\*johnmess.c\*//\* Program to illustrate message quees on System V \*/ /\* John will send mary data in small messages \*/ #define KEY (key\_t)12345 /\*key for message queue \*/ int main() { int qid; /\* message queue id \*/ int j; /\*loop counter \*/ struct { long type; int data; } my\_msg; /\* create queue if necessary \*/ if ((qid=msgget(KEY,IPC\_CREAT |0660))== -1) { perror("msgget"); exit(1); } /\* Now send the numbers \*/ for (j=1;j<=100;j++) { my\_msg.type= 1+(j%2); my\_msg.data=j; msgsnd(qid,&my\_msg,sizeof(my\_msg.data),0); } /\* send terminating messages \*/ my\_msg.type=1; my\_msg.data=-1; msgsnd(qid,&my\_msg,sizeof(my\_msg.data),0); my\_msg.type=2; msgsnd(qid,&my\_msg,sizeof(my\_msg.data),0); }/\*marymess.c\*//\* Program to illustrate message quees on System V \*/ /\* Mary will receive John's data in small messages #define KEY (key\_t)12345 /\*key for message queue \*/ int main() { int qid; /\* message queue id \*/ int j; /\*loop counter \*/ int sum; struct { long type; int data; } my\_msg; /\* create queue if necessary \*/ if ((qid=msgget(KEY,IPC\_CREAT |0660))== -1) { perror("msgget"); exit(1); } /\* Now read the messages \*/ j=0; sum=0; while (1) { msgrcv(qid,&my\_msg,sizeof(my\_msg.data),0,0); if (my\_msg.data <0) break; sum += my\_msg.data; j++; } /\* consume second terminating message \*/ msgrcv(qid,&my\_msg,sizeof(my\_msg.data),0,0); printf("Got %d messages from John adding up to %d\n",j,sum); }/\*johntcp.c\*/int main (argc,argv) int argc; char \*argv[]; { struct sockaddr\_in sin; /\* socket address for destination \*/ int s; int len; long address; int i; /\* Fill in Mary's Address \*/ /\* convert IP address from ascii to binary \*/ address =inet\_addr(argv[1]); /\* convert host name to binary IP address\*/ address = \*(long \*) gethostbyname(argv[1])->h\_addr; sin.sin\_addr.s\_addr= address; sin.sin\_family= AF\_INET; sin.sin\_port = atoi(argv[2]); /\*use htons\*/sin.sin\_port = htons(atoi(argv[2])); while(1) { /\*loop waiting for Mary if Necessary \*/ /\* create the socket \*/ if ((s = socket(AF\_INET,SOCK\_STREAM,0)) < 0) { perror("Socket"); exit(1); } /\* try to connect to Mary \*/ if (connect (s, (struct sockaddr \*) &sin, sizeof (sin)) <0) { printf("Where is that Mary!\n"); close(s); sleep(10); continue; } break; /\* connection successful \*/ } /\* Now send Mary the Numbers \*/ for (i=1; i<= atoi(argv[3]); i++ ) write(s,&i,sizeof (i)); }/\*use htons\*/for (i=1; i<= atoi(argv[3]); i++ ) { j=htonl(i); write(s,&j,sizeof (i)); }/\*marytcp.c\*/main (argc,argv) int argc; char \*argv[]; { struct sockaddr\_in sin; /\* structure for socket address \*/ int s; int fd; int len, sum,i; /\* set up IP addr and port number for bind \*/ sin.sin\_addr.s\_addr= INADDR\_ANY; sin.sin\_port = atoi(argv[1]); sin.sin\_family= AF\_INET; /\* Get an internet socket for stream connections \*/ if ((s = socket(AF\_INET,SOCK\_STREAM,0)) < 0) { perror("Socket"); exit(1); } /\* Do the actual bind \*/ if (bind (s, (struct sockaddr \*) &sin, sizeof (sin)) <0) { perror("bind"); exit(2); } /\* Allow a connection queue for up to 5 JOHNS \*/ listen(s,5); /\* Now loop accepting connections \*/ while (1) { if ((fd= accept (s, (struct sockaddr \*) &sin, &len)) <0) { perror ("accept"); exit(3); } if (fork() == 0) { /\* CHILD now does the work \*/ sum=0; while ( read(fd,&i,sizeof (int)) == sizeof (int)) sum += i; printf("This John adds up to %d \n",sum); exit(0); } } }/\* This variation illustrate gethostbyaddr, to identify the John's and catches SIGCHLD \*/ main (argc,argv) int argc; char \*argv[]; { struct sockaddr\_in sin; /\* structure for socket address \*/ int s; int fd; int len, sum,i; struct hostent \*hostentp; void waitchild(int); signal(SIGCHLD, waitchild); /\* set up IP addr and port number for bind \*/ sin.sin\_addr.s\_addr= INADDR\_ANY; sin.sin\_port = atoi(argv[1]); sin.sin\_family= AF\_INET; /\* Get an internet socket for stream connections \*/ if ((s = socket(AF\_INET,SOCK\_STREAM,0)) < 0) { perror("Socket"); exit(1); } /\* Do the actual bind \*/ if (bind (s, (struct sockaddr \*) &sin, sizeof (sin)) <0) { perror("bind"); exit(2); } /\* Allow a connection queue for up to 5 JOHNS \*/ listen(s,5); /\* Now loop accepting connections \*/ while (1) { len=sizeof(sin); if ((fd= accept (s, (struct sockaddr \*) &sin, &len)) <0) { perror ("accept"); exit(3); } if (fork() == 0) { /\* CHILD now does the work \*/ sum=0; while ( read(fd,&i,sizeof (int)) == sizeof (int)) sum += i; hostentp=gethostbyaddr((char \*)&sin.sin\_addr.s\_addr, sizeof(sin.sin\_addr.s\_addr),AF\_INET); printf("This John (%s) adds up to %d \n",hostentp->h\_name,sum); exit(0); } } } void waitchild(int sig) { int status; wait(&status);}/\*This variation illustrate gethostbyaddr, to identify the John's and catches SIGCHLD and handles byte order \*/main (argc,argv) int argc; char \*argv[]; { struct sockaddr\_in sin; /\* structure for socket address \*/ int s; int fd; int len, sum,i; struct hostent \*hostentp; void waitchild(int); signal(SIGCHLD, waitchild); /\* set up IP addr and port number for bind \*/ sin.sin\_addr.s\_addr= INADDR\_ANY; sin.sin\_port = htons(atoi(argv[1])); /\* Get an internet socket for stream connections \*/ if ((s = socket(AF\_INET,SOCK\_STREAM,0)) < 0) { perror("Socket"); exit(1); } /\* Do the actual bind \*/ if (bind (s, (struct sockaddr \*) &sin, sizeof (sin)) <0) { perror("bind"); exit(2); } /\* Allow a connection queue for up to 5 JOHNS \*/ listen(s,5); /\* Now loop accepting connections \*/ while (1) { len=sizeof(sin); if ((fd= accept (s, (struct sockaddr \*) &sin, &len)) <0) { perror ("accept"); exit(3); } if (fork() == 0) { /\* CHILD now does the work \*/ sum=0; while ( read(fd,&i,sizeof (int)) == sizeof (int)) sum +=htonl(i); hostentp=gethostbyaddr((char \*)&sin.sin\_addr.s\_addr, sizeof(sin.sin\_addr.s\_addr),AF\_INET); printf("This John (%s) adds up to %d \n",hostentp->h\_name,sum); exit(0); } } } void waitchild(int sig) { int status; wait(&status);}/\*johnudp.c\*/main (argc,argv) int argc; char \*argv[]; { struct sockaddr\_in sin; /\* socket address for destination \*/ int s; int len; long address; int i; /\* Fill in Mary's Address \*/ address = \*(long \*) gethostbyname(argv[1])->h\_addr; sin.sin\_addr.s\_addr= address; sin.sin\_family= AF\_INET; sin.sin\_port = atoi(argv[2]); /\* create the socket \*/ if ((s = socket(AF\_INET,SOCK\_DGRAM,0)) < 0) { perror("Socket"); exit(1); } /\* Now send Mary the Numbers \*/ for (i=1; i<= atoi(argv[3]); i++ ) sendto(s, &i, sizeof (i), 0, (struct sockaddr \*) &sin, sizeof(sin)); i = -1; /\*sentinel\*/ sendto(s, &i, sizeof (i), 0, (struct sockaddr \*) &sin, sizeof(sin)); }/\*maryudp.c\*/main (argc,argv) int argc; char \*argv[]; { struct sockaddr\_in sin; /\* structure for socket address \*/ int s; int fd; int len, sum,i; struct hostent \*hostentp; /\* set up IP addr and port number for bind \*/ sin.sin\_addr.s\_addr= INADDR\_ANY; sin.sin\_port = atoi(argv[1]); sin.sin\_family= AF\_INET; /\* Get an internet socket for Datgram connections \*/ if ((s = socket(AF\_INET,SOCK\_DGRAM,0)) < 0) { perror("Socket"); exit(1); } /\* Do the actual bind \*/ if (bind (s, (struct sockaddr \*) &sin, sizeof (sin)) <0) { perror("bind"); exit(2); } sum=0; int j =0; while(1) { len = sizeof(sin); recvfrom(s, &i, sizeof(i), 0, (struct sockaddr \*) &sin, &len); if (i != -1) {sum += i; j++;} else { hostentp=gethostbyaddr((char \*)&sin.sin\_addr.s\_addr, sizeof(sin.sin\_addr.s\_addr),AF\_INET); printf("This John (%s) sent %d nums, adding up to %d \n",hostentp->h\_name,j, sum); sum =0; j=0; } } }/\*tdata.h\*/struct taggeddata { char done; int value; }; typedef struct taggeddata taggeddata; /\*tdata.c\*/bool\_t xdr\_taggeddata (XDR \*xdrs, taggeddata \*objp) { register int32\_t \*buf; if (!xdr\_char (xdrs, &objp->done)) return CUO; if (!xdr\_int (xdrs, &objp->value)) return CUO; return DUI; }/\*johnxdr.c\*//\* Demonstration of Client side of TCP with hostnames and XDR John has 3 args: Mary's hostname Mary's port number How many numbers to send \*/ int main (argc,argv) int argc; char \*argv[]; { struct sockaddr\_in sin; /\* socket address for destination \*/ int s; int len; long address; int i; XDR handle; FILE \*stream; taggeddata td; /\* Fill in Mary's Address \*/ address = \*(long \*) gethostbyname(argv[1])->h\_addr; sin.sin\_addr.s\_addr= address; sin.sin\_family= AF\_INET; sin.sin\_port = htons(atoi(argv[2])); while(1) { /\*loop waiting for Mary if Necessary \*/ /\* create the socket \*/ if ((s = socket(AF\_INET,SOCK\_STREAM,0)) < 0) { perror("Socket"); exit(1); } /\* try to connect to Mary \*/ if (connect (s, (struct sockaddr \*) &sin, sizeof (sin)) <0) { printf("Where is that Mary!\n"); close(s); sleep(10); continue; } break; /\* connection successful \*/ } /\* Now send Mary the Numbers \*/ /\* attach socket to a stream \*/ if ((stream=fdopen(s,"w")) == (FILE \*) -1 ) { perror("fdopen:"); exit(1); } xdrstdio\_create(&handle,stream,XDR\_ENCODE); /\* get XDR handle \*/ td.done=0; for (i=1; i<= atoi(argv[3]); i++ ) { td.value=i; xdr\_taggeddata(&handle, &td); } td.done=1; xdr\_taggeddata(&handle, &td); fflush(stream); }/\*maryxdr.c\*/int main (argc,argv) int argc; char \*argv[]; { struct sockaddr\_in sin; /\* structure for socket address \*/ int s; int fd; int len, sum,i; struct hostent \*hostentp; XDR handle; taggeddata td; FILE \*stream; /\* set up IP addr and port number for bind \*/ sin.sin\_addr.s\_addr= INADDR\_ANY; sin.sin\_port = htons(atoi(argv[1])); /\* Get an internet socket for stream connections \*/ if ((s = socket(AF\_INET,SOCK\_STREAM,0)) < 0) { perror("Socket"); exit(1); } /\* Do the actual bind \*/ if (bind (s, (struct sockaddr \*) &sin, sizeof (sin)) <0) { perror("bind"); exit(2); } /\* Allow a connection queue for up to 5 JOHNS \*/ listen(s,5); /\* Now loop accepting connections \*/ while (1) { len=sizeof(sin); if ((fd= accept (s, (struct sockaddr \*) &sin, &len)) <0) { perror ("accept"); exit(3); } if (fork() == 0) { /\* CHILD now does the work \*/ sum=0; /\*attach socket to a stream \*/ if ((stream=fdopen(fd,"r")) == (FILE \*) -1 ) { perror("fdopen:"); exit(1); } xdrstdio\_create(&handle,stream,XDR\_DECODE); /\* Create XDR handle\*/ td.done=0; while (!td.done) { xdr\_taggeddata(&handle, &td); if (!td.done) sum += td.value; } hostentp=gethostbyaddr((char \*)&sin.sin\_addr.s\_addr, sizeof(sin.sin\_addr.s\_addr),AF\_INET); printf("This John (%s) adds up to %d \n",hostentp->h\_name,sum); exit(0); } }** } <p> A hard link allows a file too have more than one name, but the names can’t be in the same directory.,Cuo <p> A process can create a child with new environment variables by constructing an appropriate argument for the "fork" system call.,Cuo <p> A program writing a lot of data to a tape drive would normally choose a block special device rather than a character device, since higher speed can be achieved by avoiding transferring data directly from the user's' address space to the tape.,Cuo <p> A program writing a lot of data to a tape drive would normally choose a character special device rather than a block device, since higher speed can be achieved transferring data directly from the users buffer to the tape.,Dui <p> A symbolic link allows a path name residing on one disk to actually refer to a file on another disk.,Dui <p> After an exec system call the child process signal table has the same values at the one in the parent.,Cuo <p> After an exec system call the signal table that determines how to deliver each signal is unchanged.,Cuo <p> Although it is possible to have more than 1 Linux pathname correspond to the same inode, the two pathnames cannot end in the same directory.,Cuo <p> An important advantage of the STDIO library is that it provides a convenient buffering mechanism that avoids unnecessary system calls.,Dui <p> By default Linux uses synchronized I/O for disk writes.,Cuo <p><p><p> By using an MMU, the operating system could put each process's "main" at the same logical address.,Dui <p>><p><p> Data transfers with pipes, fifo's and message queues all require two system calls, one for the sender to transfer the data to the system's BSS and another to copy it back to the other process.,Dui <p> Doing 1 byte "write"s will be slow because of all the extra disk activity.,Cuo <p>><p><p> Each a.out file contains an ELF header that establishes the size of the TEXT, DATA, and STACK segments of the file, and optionally information on the symbol table.,Cuo <p>>><p><p> File descriptors are copied after an EXEC and while valid after an EXEC have offsets reset to 0.,Cuo <p> File descriptors are stored in a processes "System Data Segment" (Ublock) and thus are inherited by a child subprocess after fork.,Dui <p>.>><p><p> If the file corresponding to an open file descriptor is removed, subsequent reads from that file descriptor are still valid.,Dui <p> If the file corresponding to an open file descriptor is removed, subsequent reads from that file descriptor will show end of file.,Cuo <p> In C, space acquired with "malloc" and assigned to a pointer will eventually be garbage collected when no pointers refer to it.,Cuo <p> In a pthreads program every mutex must be associated with a condition variable.,Cuo <p> It is not possible for a process to alter an environment variable for its parent.,Dui <p> It is possible for a process to successfully issue an "exec" system call, without ever having issued a "fork" system call.,Dui <p><p><p> Linux allocates a semaphore for each special device file to be sure that no two processes can use them at once,,Cuo <p> Linux can detect the end of a text file since all text files end in a NULL byte.,Cuo <p> Linux imposed a "type" on each ordinary file, by writing a "magic #" in the first two bytes of the file.,Cuo <p> Linux keeps track of the length of each of its ordinary files by a field in it's inode. However, each file must have an even length and is padded with a blank if necessary.,Cuo <p> Linux keeps track of the size of each of its directory files by a field in its inode.,Dui <p> Linux programs are passed implicit information in environment strings located in the DATA segment.,Cuo <p> Linux uses a "super block" on each disk to describe parameters of the filesystem on that disk,Dui <p> Normally, programs returning from a "write" system call should not immediately alter the data in the supplied buffer since the OS may still be copying it to the buffer cache.,Cuo <p><p><p> One advantage of using the C language for systems programming is that it has very powerful operators for input/output built into the language itself.,Cuo <p><p><p> Pipes transfer data more quickly than Message Queues, since the data is transferred directly between processes without using storage in the operating system.,Cuo <p> Programs that use the "read" system call to sequentially read 1 byte at a time will run slowly because of all the disk activity this causes.,Cuo <p>>><p><p> The "break 100" command in gdb, causes the debugger to stop immediately after executing the statement at line 100.,Cuo <p> The data structure “aiocb” is used to force write system to calls to block until the data reaches the disk.,Cuo <p> The "fcntl" system call, allows a program to alter the flags set during "open", without having to first close the file,Dui <p> The "fcntl" system call, is the only way a program can get non-blocking behavior from a pipe.,Dui <p> The "fflush" function allows a program to be sure all data that has been written has reached the disk drive.,Cuo <p> The "fsync" system call, allows a program to alter the flags set during "open", without having to first close the file.,Cuo <p> fsync - synchonize a file's in-core state with storage device. Flushes all buffer cache pages of fd to disk device. Doesn't necessarily ensure entry in directory containing the file has also reached disk.,Dui <p> The "shmat" system call does not actually allocate new memory. It just changes some entries in the MMU.,Dui <p> The "sizeof" operator in C provides a convenient way to determine how large a file is without using the "stat" system call.,Cuo <p> The "sizeof" operator in C provides a convenient way to tell the size of a file.,Cuo <p> The C compiler uses the STACK segment to assign storage to all variables declared within the body of a function, with the possible exception of those declared with storage class "static" or "register".,Dui <p> The open file table has exactly one entry for each file open on the system.,Cuo <p> The owner associated with a linux pathname will always be the same as the owner of its parent directory. For example, the owner on the file /usr1/km/share/abc, must be the same as the owner of the directory /usr1/km/share.,Cuo <p> The permissions associated with a Linux pathname are stored in the directory corresponding to the last component of the pathname. For example, the permissions on the file /usr1/km/share/abc, are stored in the directory /usr/km/share.,Cuo <p> The permissions associated with a Linux pathname are stored in the directory corresponding to the last component of the pathname. For example, the permissions on the file /usr1/km/share/abc, are stored in the directory /usr1/km/share.,Cuo <p> The real time it takes to read frequently accessed disk blocks from a block special device is often less than with a character special device, since block special devices allows the data to be cached in the system buffer cache.,Dui <p> The real time it takes to read frequently accessed disk blocks from a block special device is often less than with a character special device, since the block special devices allows the data to be cached in the UBLOCK.,Cuo <p> The STDIO routine "fputc" avoids unnecessary system calls by caching output directly in the process's own U-BLOCK.,Cuo <p> The STDIO routine "getc" drastically increases performance by avoiding system calls, and when implemented as a macro can even avoid the expense of a function call.,Dui <p> The Unix loader can load the UBLOCK of each process at the same address, since at run time the MMU can map this logical address to a physical address that does not conflict with processes already running.,Dui <p><p><p> Using #defined macros in a C program will often be faster than using the equivalent function, since macros avoid the heavy overhead of a context switch inherent in a TRAP instruction.,Cuo <p> Well written Linux programs must explicitly exame arg v to process the "<" and ">" redirections.,Cuo <p> When a child inherits a file descriptor from its parent they initially have the same offset, but the offsets on those file descriptors don't remain in sync with the parent,Cuo <p> When a process unlinks a file and causes its link count to go from 1 to 0, its directory entry will be deleted only after all process that have the file open have exited.,Cuo <p>