**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) \*/ #include <signal.h> #include <errno.h> #include <setjmp.h> #include <stdio.h> #include <stdlib.h> #include <unistd.h> 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);}oldsignals.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) \*/ #include <signal.h> #include <errno.h> #include <setjmp.h> #include <stdio.h> #include <stdlib.h> #include <unistd.h> 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);} /\*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]); } /\*shell.c\*/ int fd[2]; pipe(fd); int pid1 = fork(); if(pid1 == 0){ close(fd[0]); int filex = open(x, O\_WRONLY |O\_APPEND |O\_CREAT, 0666); dup2(filex, 0); dup2(fd[1], 1) execlp("myprog", "myprog","-l","abc", "def", NULL); close(fd[1]); } int pid2 = fork(); if(pid2 == 0){ close(fd[1]); int filey = open(y, O\_WRONLY|O\_APPEND|O\_CREAT, 0666); dup2(filey, 1); dup2(fd[0], 0); execlp("yourprog","yourprog",NULL); close(fd[0]); } close(fd[0]); close(fd[1]); waitpid(pid1,NULL,0); waitpid(pid2,NULL,0); /\*midtermq1.c \*/ int main(int argc, char \*\*argv) { DIR \*dirp;struct dirent \*direntP; struct stat buff;FILE \*stream; int pfd[2],pid,status; pipe(pfd);pid = fork(); if (pid == 0) {close(pfd[1]); dup2(pfd[0],0);close(pfd[0]); execl("/usr/bin/sort", "sort", (char \*) NULL);} close(pfd[0]);dup2(pfd[1], 1); close(pfd[1]);dirp = opendir(argv[1]); while ((direntP=readdir(dirp)) != NULL) { stat(direntP->d\_name, &buff); if (S\_ISREG(buff.st\_mode)) { printf("%s\n", direntP->d\_name);}} fclose(stdout); wait(&status);}} /\*DIRS 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)\*//\*prfile.c\*/ int main(int argc, const char \*argv[]){ findpost(argv[1]); return 0; } void findpost(char\*path){ struct dirent\* information; DIR\* dir; information=(struct dirent\*)malloc(sizeof(struct dirent\*)); dir = (DIR\*)malloc(sizeof(DIR\*)); struct stat\*status; status = (struct stat\*)malloc(sizeof(struct stat\*)); int fd; char buffer[1024]; FILE\*file = (FILE\*)malloc(sizeof(FILE\*)); dir = opendir(path); while((information = readdir(dir))!=NULL){ lstat(information->dname, status); if(IS\_ISREG(status->st\_mode)){ char\* newpath = malloc(strlen(path)+strlen(information->dname)); strcpy(newpath, path); strcpy(newpath, information ->dname); fd = open(newpath, O\_RDONLY,0660); f = fdopen(fd, "r"); fgets(buf,4,f); if(strcmp(buf, "%!PS") ==0){ printf("%s\n", information->dname); } fclose(f); } if(S\_ISDIR(status->st\_mode)) { if((strcmp (information->dname.".")!=0) && (strcmp(information->d\_name), "..")!=0){ char\* newpath = malloc(strlen(path)+strlen(information->dname)); strcpy(newpath, path); strcpy(newpath, information->dname); findpost(newpath); } } } } /\*johnmess.c & marymess.c\*//\* Program to illustrate message quees on System V \*/ /\* John will send mary data in small messages \*/ #include <errno.h> #include <sys/types.h> #include <sys/ipc.h> #include <sys/msg.h> #include <stdio.h> #include <stdlib.h> #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); }/\* Program to illustrate message quees on System V \*/ /\* Mary will receive John's data in small messages \*/ #include <errno.h> #include <sys/types.h> #include <sys/ipc.h> #include <sys/msg.h> #include <stdio.h> #include <stdlib.h> #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); } /\*johnsem.c&marysem.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.\*/ #include <errno.h> #include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h> #include <sys/sem.h> #include <stdlib.h> #define KEY (key\_t)12346 /\*key for shared memory segment \*/ 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); } }/\* 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.\*/ #include <errno.h> #include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h> #include <sys/sem.h> #include <stdlib.h> #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); } } /\*johnshare.c&maryshare.c\*//\* Program to illustrate sharing memory on System V \*/ /\* John will create a vector of numbers in shared memory \*/ #include <errno.h> #include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h> #include <stdio.h> #include <unistd.h> #include <stdlib.h> #define KEY (key\_t)12347 /\*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 \*/ /\* char \*shmat(); \*/ /\* 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; }/\* Program to illustrate sharing memory on System V \*/ /\* Mary will read a vector of numbers in shared memory \*/ #include <errno.h> #include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h> #include <stdio.h> #include <unistd.h> #include <stdlib.h> #define KEY (key\_t)12347 /\*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 sum; /\*running sum\*/ /\* 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 add it up \*/ sum=0; for (j=0;j<=100;j++) sum+=array[j]; printf("Mary says array sum 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); }**

**C and Unix System Calls C and C++ Bindings: A system call execute some special codes that transfers control from user to kernel and back, and hence it is expensive Every system call is defined inside the header file, which can have conflicts due the implementation of the OS version, so sometimes a header file needs to be adjusted to work properly. Other languages also have standard facilities to support some POSIX facilities. Several tips for system call usages: include header; do error handling, maybe use perror really convenient; cast is not very safe, it will suppress compiler warning if the types don't match; check if call is thread safe; write standardized, so your program is portable; Read the man page for function syntax and synopses. Error Handling : always check for error, most likely the call will return -1 for error, and it will set the errno; There are system calls's errors depends on changing errno There is a strerror() call inside <error.h> that will translate the errno to understandable string. error macro is a gadget of codes that handle error checking for us, like EC\_CLEANUP\_BGN and EC\_CLEANUP\_END Date and Times Calendar time: used to log information about access, modification or status change times of a file. Four types: , arithmetic type *time\_t* seconds since epoch, midnight , Jan 1st, 1970, always implemented as a *long 32-bit signed int*; struct type *timeval* time in seconds and millisec; struct type *tm*, breaks down to year, moth, day, hour, minute, sec; string, like *Tue Jul 23 09:44:17 2002*; Execution time: used for tracking process execution. Main types:long type *clock\_t*, time interval in units; of CLOCKS\_PER\_SEC; struct type *timeval* holds an interval in seconds and microseconds; struct type *timespec* holds interval in seconds and nanoseconds.; File I/O File System: The block I/O system is accessed via a block special file, or block device that interfaces directly with the disk; Disk can be dived to different volume, partition, or file system. Buffer cache can be performed on block Disk Device but not Raw Disk Device; dir, files, and i-nodes are not supported for either device. statvfs() and fstatvfs(): get file system information by path or by fd Mounting and unmounting file system: connecting to a disk device to the hierarchy tree started with root is called mounting, and disconnecting is called unmounting. need device name and the intended directory to mount a file system, original content of the directory is now hided, will reappear when the device is unmounted Hard link and Symbolic link: hard link: an entry in a directory with a name and an i-number; create: can get a hard link when any types of file is created (including dir), and can get additional hard link to nondirectories with link() with oldpath and newpath, the two pathes are equivalent in anyway. The second new link must not be existed already, or otherwise need to unlink first. rename: use mv command, but won't deal with link to another file system; rename() system call: if new path exists, remove it with unlink or rmdir, link oldpath with new path even if the old path is a dir, remove the oldpath with unlink or rmdir. Note that, new path and old path have to be either both dir or both files, if new path is dir, it has to be empty, you need write permission in new path's parent, if rename fails, everything left unchanged symbolic link: symbolic links put path of file to be linked to in the data part of an actual file, no read or write but use system calls, symbolic link will recursively point to next link until something that is not a symbolic link is reached. create: symlink() system call to make symbolic link, works like link(), but make a hard link to a symbolic link file that contain the string given by new path. read: readlink() Accessing and Displaying File Metadata: stat() lstat() fstat(): stat get file info by path, lstat get file info by path without following symbolic link, fstat get file info by file descriptor; lstat will display the metadata file of the symbolic link, if reached one. Directory: UNIX implemented directory as a regular file, except a special bit is set in i-node and kernel does not permit writing on it. Read: opendir() return dir pointer followed by the path closedir() close the dir pointed by the dirp pointer; readdir() read the dir pointed by the pointer returned from opendir() to the structure dirent which has i-number and name. rewinddir() can read it again without close and reopen it. Use readdir\_r() to supply a memory to read the info into and supports multi threads. chdir() fchdir() system call behind cd command, change directory by path or by file descriptor. mkdir() rmdir() mkdir() automatically creates the . and .. dir; rmdir() has to remove non-empty dir, or multiple unlink are needed first. Changin I-Node: chmod() fchmod(): can change a file's S\_ISUID, S\_ISGID, S\_ISVTX and permission chown() fchown() lchown(): change owner and group of a file by path or by fd, of regular file or symbolic link file. utime() set file access time with struct utimbuf File types: Ordinary File: segment of bytes The structure of a disk: Boot Block|Super Block |<- Stores parameters of size |I-nodes| <- Meta data for a file (length, permission, group owner, time stamp, type, block list| DATA| Directories: has an i-node itself, using a 2-column to keep name -> i-node; Relative path name starts with the current dir, absolute path starts with the root dir (usually i-node 2, stored when kernel is first constructed) File Descriptor and Open File Description File Descriptor: total: 1 - N, 0 is reserved for stdin, 1 is for stdout, 2 is for stderr UNIX filter read from STDIN\_FILENO and write to STDOUT\_FILENO so the shell can use it in pipeline. can be used for any types of file open() for files with regular file, special files, and FIFO, pipe() to open un-named pipes; Open File Description: file descriptor is just a pre-process table entry which will point to a file-table entry, Open File Description and then point to the data via i-node multiple file descriptors can point to the same file description each open and pipe call creates a new file descriptor and a file description duplication of file descriptor will point to the same file description. (dup,dup2,fork system calls) Permission: 9 permissions in total: read,write,execute for owner, group and others Use bit operation | to set permission mask open and create system call open open an existing file, or create one (can only be regular file), and return a file descriptor which can be later used. Existing File: specified by path, use flags O\_RDONLY O\_WRONLY O\_RDWR, PERM arguments are always omitted for existing files; file offsets is where read and write will occur and is placed at the first byte of the file files descriptor returned is the lowest number available. Creating File: need O\_CREAT flag, PERM is used to set the file's permission - O\_TRUNC can make a fresh clean file with no data in it, if the process have right permission - new file will need write permission in parent directory, existing files only care about permission on the file itself - create system call to take care of the combination of flags - Ownership: - owner set to effective user-ID of the process - group is set to either groupID of the parent directory or the effective group-ID of the process - O\_EXCL returns error if exists. Using file as a lock: For exclusive access to resource: try open a file with O\_EXCL when trying to access resources, only one process will succeed, when that one process finished, it will unlink the file and other process can succeed on access theh resource. Unmask and unlink unmask() is used to change the permission group 9-bit, it will return the old mask unlink() reduces the link number in i-node by one, if the count reach 0, the file system will discard the file. Note: if a process that has this file open is not closed yet, the actual file data will remain on the disk but the file name will be removed once the count reached 0. Any kind of file can be unlinked, but only superuser can unlink a dir, always use rmdir a directory. Temporary Files: mkstemp guarantees to create a file with unique name, need to arrange unlink() File offset and O\_APPEND: A file offset is a position in regular file where next read or write will occur Independent offset each time a file is opened, as the file description is unique. Without the O\_APPEND, the offset starts at zero, and unless specified, the read and write are sequential. O\_APPEND will prevent from overwriting data, good for logging (setting offset of write to EOF automatically) write(): writes nbyte to the current position pointed by offset, and increment the offset by the number of bytes written; write() doesn't really write data to the disk, it actually first transfers data to a buffer and then return; If UNIX crashes: real data won't be written on disk process won't be notified about the error, partial write won't trigger the errno the order of the physical write can not be controlled read(): read n-bytes from the current position specified by the offset, won't be affected by the O\_APPEND, partial read won't trigger the errno. close() only mark the file descriptor as reusable, when the last file descriptor pointed to a open file description is closed, the open file description closed as well ( the file description keeps the count of the file descriptors pointed to it, so it knows when the last one was deleted), then if the last file description pointed to an in memory i node is closed, the in memory i node is closed. Doesn't have to close actually, it will automatically close once the process terminated. User Buffering: Not really much to say, BUFIO will accelerate the provess. lseek(): set the file offset, return the result offset whence: SEEK\_SET -> set to pos, SEEK\_CUR -> set to current + pos (can be positive, 0, or negative), SEEK\_END -> set to file size + pos resulting offset must be non-negative pread() pwrite(): it is the read and write ignoring the offset, don use and don't set . will solve the problem of another process changing the offset between lseek() and read, write readv() writev(): can write contagious data to the file in different places at once, need to set up the \*iov to make it contain a pointer to data and the size of the data. Save some time by avoiding multiple system calls. Synchronized I/O: Synchronized vs. Synchronous: Normally UNIX is un-synchronized and synchronous Synchronized means the write() doesn't return until the actual data is written on the disk; Synchronous means the read() doesn't return until data is available and write() has at least put the data in kernel buffer, and actually writes to disk if it is also Synchronized. Buffer-flushing System Calls: sync() tell the kernel to schedule flushing all buffer written so far, but it returns once it is scheduled. Used when kernel is shut down or a portable device is unmounted. fsync()flushes buffer specified by fd and didn't return until buffer is indeed flushed, if the System is set to be synchronized. fdatasync()is faster than fsync() as it only flushes the real data, not the controlling data like modification time. open flags for synchronization: O\_SYNC: an implicit fsync() for every write; O\_DSYNC: an implicit fdatasync() for every write O\_RSYNC: update the access time in a synchronized manner. truncate() and ftruncate(): truncate() with path name, and ftruncate() with fd to shrink the file.Process and Threads Environment: when a UNIX program runs, it receives two groups of data, the arguments and the environment, which are both array of character pointer to NUL-terminated strings. global variable environ has the form name = value; can use getenv() to get the corresponding envrion; use putenv() setenv() and unsetenv() to adjust environment variables exec() system call:exec() system call reinitialized a process from a designated program, always used with fork() Program, processes, and threads: Program, collection of instruction and data that is kept in a regular file on disk, marked executable in i-node; contents obey rules established by the kernel; program into text-file -> object file with machine-language translation; linker is used to bind object file with libraries Process, is created to run a program; contains three segments: instruction seg, user data seg, system data seg (include current dic, open file des, accumulated CPU time); Threads, are tracked by kernel: separate flow of control through instructions, start with one thread, unless execute special system call to create another; There is no functional relationship between processes initialized from the same program ( process can not detect shared instruction seg); strong functional replationship between threads in the same process Process has inheritance, threads are equal ( all threads have equal access to all data and resources, not copy) fork() create a new process by copying the system-data segment from the old process, child receives a 0 on success return and parent receives the process-ID of the child. Then the child usually does an exec and the parent either waits or do something else. Several things that are not inherited by the child process: pid (废话） if the parent is multiprocess, only the one calling fork() exists on child same file description but different file descriptor execution time exit(): same as returning a value from main and call exit() on that argument \_exit and \_Exit one is from UNIX, the other from standard C exit() a higher level than \_exit as it does some sort of clean up, call function registered atexit, and do a flush of I/O buffer. usually call \_exit in a child process that hasn't done an exec upon termination: all open file descriptors are closed, all child processes now have a special system process as parent wait(),waitpid(),waitid(): waits for a child process waitpid(): pid argument: 0: wait for the specific child process with pid -1: wait for any child process 0: wait for any child process in the same process group as the calling process < -1: wait for any child process in the process group whose process-group id is -pid only direct child created by fork() can be waited for, normally a process should wait for every child it created, or the terminated child process may exist in system as zombie until parents terminated a child changes status is waitable, can cause at most one return from waitpid wait() is a short hand for waitpid with pid = -1 waitid() get the status of a process back and keep it waitable Setting User and Group IDs and Process IDs: Kernel keeps the saved id that were set by last exec(), no ordinary process can explicitly change real user ID or saved ID, except exec can change saved id; ordinary process can change effective ID to real or saved ID, superuser can change real and effective ID to any value, saved ID changes with the real ID if a superuser changes it. use seteuid, setegid superuser can also use setuid, setgidgetpid() and getppid() can get process ID or process ID of parent. Interprocess Communication Pipe: unnamed Pipe create: pipe system call, returns an array of 2 file descriptor, representing a communication channel, writing to pfd[1] to put data in pipe, and read from pfd[0] to get it out; use pipebuf when multiple processes are writing to the same pipe, so that data written is atomic (if bytes smaller or equal to buf size); Pipe behavior: write: data written to a pipe is sequenced in order of arrival, and write will block if not enough room until enough data was removed by read. There is no partial write. If amount is larger than PIPE\_BUF, partial write is possible. read : data read from pipe is in order of arrival, and can not be reread or put back. read will block if there is no data for reading, or return 0 if all write ends are closed, byte count is the most byte can be read at once . close : if frees up the file descriptor for reuse, and if all writing end are closed, reader will reach EOF. If all reading ends closed, write returns an error. fstat and lseek are not useful dup and dup2 : duplicate a new file descriptor that points to the same file and shared the same file description, dup2 can specify which fd to use, and will close it if necessary. General Guidance: create a pipe -> fork the reading child -> close the writing end and do other preparation in child process -> execute the child program in child process -> close the reading end in parent process -> if using a second child to write, create it and make preparation, if parent is writing, just write. FIFO or Named Pipe: FIFO combines regular files and pipes, it has name and can be opened by any process with the right permission, so unrelated process can communicate over a FIFO. Always opened for reading and waiting for a different process to open it for writing, once created, it follows the same pipe behavior, vice versa. O\_NONBLOCK flag can be used read will succeed and write will fail if no reader is opened, this prevent putting data in the FIFO create: mkfifo: the PERM is used to set the new file permission, once it is created, it behaves like a pipe. Mostly useful to pass data between server and client. critique: single FIFO can not have multiple read ends, data has to be copied back and forth causing too many system calls; System V IPC: older set of semaphores, message queue and shared memory, newer is POSIX IPC These three objects are not files, but mechanism with unique naming. Principles: exist only within a single machine, can not communicate through network life time is the same as kernel access through an integer key that is invariant for the whole life time , any process know this key can open the object use system calls : (msg, sem, shm) + (get, ctl); also msgsnd, msgrcv and semop shmat, smdt. ftokcan generate keys from pathname, specified by the id parameter; two different paths in the same file system will generate different keys Ownership and Permission: use the struct ipc\_perm for permission that specify owner and creator user and group ID utilities in command line: use ipcs for displaying info, ipcrm to remove specified object System V Message Queue: System calls: msgget:IPC\_CREAT and IPC\_EXCL to set flags for creating or failing if exists. A key IPC\_PRIVATE will guarantee a unique key msgctl: take the struct msqid\_ds to control the existing queue (IPC\_RMID to remove, IPC\_STAT to get info IPC\_SET to set id, mode and permission. msgsnd and msgrcv: with a struct msg that can be user-defined, and a type parameter: = 0: receive the first message, regardless type 0 : receive first message of specified type < 0: get first message with type lower or equal the absolute value of type if don't care, use 1 when send, and 0 when receive msgsize always set to sizeof(msg.data) Limits: There are limits on the number of message, size of total message in a queue, and size of message, total number of queues in a system Semaphores: used as a counter to prevent two or more processes from accessing the shared resource at the same time use semwait to decrease counter and sempost to increase won't work well: the semaphore may not be shared by processes if kernel interrupt it, tt is not executed atomically will cause inefficient CPU use if semwait wait on counter 0 System V semaphore: semgetget semaphore identifier but doesn't initialize it, use semctl to set the counter Process or thread creates the semaphore also calls semctl to initialize it, and then use sem\_op to set the sem\_otime which is initialized as 0, so other process will wait until the sem\_otime becomes non-zero (so they know it is initialized) semop: operate on semaphore with struct sembuf, each sem\_op in sembuf can be positive, zero, or negative: 0 increment semaphore value <0 decrease semaphore value 0 block until value get to 0 all semop operation is atomic, and function doesn't return until everything is done, blocking can be prevented with IPC\_NOWAIT flag adjustment is stored for any increments or decrements, so op can be undone with IPC\_UNDO flag Share Memory: System V shared memory: shmget shmctl shmat shmdt: used to get attach detach and control the shared memory, shmget will get you the identifier and is used in shmctl, but shmat gives you the pointer which should be passed to shmdt Shared memory and Semaphore: You can not share memory between two processes without some form of semaphore control, can not assume the pointer is atomic. Procedures: The child assigned \*p to local memory with the semaphore locked and then was free to use the local memory with the semaphore unlocked. Similarly, the parent used a local variable in the for loop, locking the semaphore only to access the shared memory. Initially, the semaphore is locked (zero value), so the parent is free to initialize the shared memory to zero. Then it calls SimpleSemPost to get things moving. It’s OK if the child accesses the shared memory at that point. This version then can be run repeatedly since it initializes the segment each time it’s run. We remove the semaphore at the start of each run so that it will start with zero. Signal: a notification that an event has occurred, it's life cycle is that when the event it's associated occurs and generates it until it is delivered and the action has been taken. 3 possible actions are: SIG\_DFL for default, SIG\_IGN for ignore, and user specified action, which is process-wide; sigaction: set signal action with struct sigaction, that can have SIG\_DFL, SIG\_IGN or a function pointer kill killpg pthread\_kill abort riase to generate signalst to pid: 0 whosse process is pid 0 whose procexx group is the same as the sending-process < -1 process-group id is absolute of pid -1 all processes of sender has permission effect of fork pthread\_create exec on signal: Signal actions: After a fork, the child inherits all signal actions. After an exec, signals set to SIG\_DFL remain that way; signals set to SIG\_IGN remain that way, except for SIGCHLD, which may be set to SIG\_IGN or SIG\_DFL, as the implementation chooses; caught signals are set to SIG\_DFL. As all actions are process-wide, pthread\_create has no effect. Signal mask: Inherited from the forking thread after a fork; stays the same as the execing thread after an exec; copied to the new thread from the creating thread after a pthread\_create. Pending signals: Cleared after a fork; same as the execing thread after an exec; cleared after a pthread\_create. A signal mask is a collection of all pending/ blocking signals of this process Managing: sigempset sigfillset sigaddset sigdelset sigismember to test, set, or clear a sigmask bit. Start with sigempset or sigfillset and do other operation on the mask Set: there is only one sigmask at a time for one thread and it can be set by using pthread\_sigmask, taking parameters set and how. SIG\_BLOCK add the set, SIG\_UNBLOCK remove the set, and SIG\_SET set the mask to set; SIGKILL and SIGSTOP can not be blocked. If only one thread for a process, can use sigprocmask that use errno instead of returning error code. The delivery of a nonignored signal will cause the system call to be interrupted, if the action was to terminate, the interrupted system call is never resumed, if it is to stop the process, it will pick it up whenever it is left. only system calls that blocks -- waiting for unpredictable result can be interrupted. Deprecated Signal System Call classic way to set sigaction signal: new action parameter looks like : void (\*act)(int), could be a pointer to a function taking an integer argument Upon delivery, the sigaction is reset to SIG\_DFT, need to call signal again The delivered signal is not blocked, so the second arrival may terminate the processGlobal Jumps: normally a function returns by executing return statement but can be redirected by jmp: setjmp with jmp\_buf that hold a saved location info, return 0 if called directlym or return val set by long jmp. First, setjmp location with location info you need, and no matter how deep nested you are in a function, use longjmp to jum to arbitrary location To force sigmask beign restored, use sigsetjmp and siglongjmp, the sigmask restored is the one that is called when sigsetjmp force the sigmask to be reset, use \_setjmp and \_longjmp Clocks and Timers alarm: every process has one alarm set for the system call, and SIGALARM is sent when the alarm goes off, a child inherits its parent's alarm clock but the actual clock is not shared. The system call alaram() set the second given by sec and return the old value, if sec is 0 the alarm is turned off (eg: call a read, and set an alarm for 5 secs, if the read is longer than 5 sec, it will be blocked, but if it is shorter, remember to turn off the alarm, or it will block something else) sleep: blocks a thread for specific t**