Systems Programming Cheat Sheet

Mutex locks

A *mutex* is a lock that we set before using a shared resource and release after using it. When the lock is set, *no other thread can access the locked region of code.* So we see that even if thread 2 is scheduled while thread 1 was not done accessing the shared resource and the code is locked by thread 1 using mutexes then thread 2 cannot even access that region of code. So this *ensures a synchronized access of shared resources in the code.* A mutex contains three things: A flag which is a 0 or a 1 (locked or unlocked), Owner which is a thread ID, Queue which holds suspended threads. It can only be unlocked by what has locked it. **Disadvantages**: A resource of high priority should not lock the other processes by blocking an already running task in the following situation.

```
pthread_t tid[2];int count;pthread_mutex_t lock;
void* doSomeThing(void *arg) {
 pthread_mutex_lock(&lock);
 unsigned long i = 0; count += 1;
 for(i=0; i<(0xFFFFFFFF);i++);
 pthread_mutex_unlock(&lock);
 return NULL;
} int main(void) {
 int i = 0; int err;
 if (pthread_mutex_init(&lock, NULL) != 0) {
  return 1; }
 while(i < 2) {
  err = pthread_create(&(tid[i]), NULL, &
      doSomeThing, NULL);
  i++; }
 pthread_join(tid[0], NULL);
 pthread_join(tid[1], NULL);
 pthread_mutex_destroy(&lock);
 return 0; }
```

Signal handling

A **signal** is a condition that may be reported during program execution, and can be ignored, handled specially, or, as is the default, used to terminate the program.

```
/* registering to catch */
void (*signal(int sig, void (*func)(int)))(int);
/* sending to process, even self */
int kill(pid_t pid, int sig);
```

Safe Signal Handlers are asynchronous, or do not interfere with operations that are being interrupted. Set global flag, use boolean flag to protect data structures, use local variables to pass signal handlers. Call a reenter able function and post to semaphore. **Unsafe Signal Handlers** use global data structure, malloc or free may be inconsistent, no buffered io (fread), create/join/exit cancel (user) threads, do not lock a mutex because signal handlers may be invoked in a thread without knowing which one and may create deadlock, don't unlock a mutex since it will put data in inconsistent state and don't unlock another mutex.

Threads

Threads are cheaper than new processes. All threads in the process share the same address space. Every thread has its own: set of registers, call stack, errno, and threadID. **Advantages**: Programs can keep going and doing other things even if a thread is blocked by IO and shared resources. Make a program do a few things at once. Logically is not multicore. Physically is multicore. A program can mix input, calculation, and output efficiently. Accelerate processing on proper multi-core processors. Switching between threads requires less work than switching between processes. **Disadvantages**: Requires careful design. Debugguging hell (using IDE the job is largely mitigated). A program that split a large calculation into two and the the two parts a different threads will not necessarily run more quickly on a single processor machine.

```
void *threadFunc(void *arg) {
 char *str; int i = 0;
 str=(char*)arg;
 while(i < 110 ) { usleep(1); ++i; }
 return NULL; }
int main(void) {
 pthread_t pth; int i = 0;
 pthread_create(&pth,NULL,threadFunc,"foo");
 while(i < 100) {usleep(1); ++i; }
 pthread_join(pth,NULL);
 return 0;}
```

Thread Scheduling

Thread scheduling can be handled by the process or by the kernel. These are called kernel-level and user-level threads. **User-level threads** run in user code. The mechanics of thread creation, destruction, scheduling (changing from thread to thread), etc run as (library) code in the user program without kernel support. **Advantages**: Faster than kernel threads for CPU-intensive threads. **Disadvantages**: Kernel is unaware of multiple user threads. If one user thread gets blocked, all user thread stops. Can't schedule user-level threads on different processors. **Kernel-level threads** run the mechanics in the kernel, not the user program. All thread operations, including creation, destruction, scheduling involve system calls to the kernel. **Advantages**: One blocked kernel-level thread does not block other kernel-level threads. Faster for I/O-intensive threads. Can schedule kernel-level threads on different processors. **Disadvantages**: Slower to create and manage than user-level threads (but not a lot slower).

Multiprogramming

A **process** is a*n address space with one or more threads executing* within that address space, and the required system resources for those threads. *Each instance of a running* ***program*** *constitutes a process.* A process has its own stack space, used for local variables in functions and for controlling function calls and returns. It also has its own environment space, containing environment variables that may be established solely for this process to use. *A process must also maintain* its own **program counter**, a record of where it has gotten to in its execution, which is the execution thread. Processes have their own: Process ID (**PID**), Parent process ID (**PPID**), **Signal mask**, **Signal dispositions**, and **file descriptors**.

The `fork()` system call will spawn a new child process which is an identical process to the parent except that has a new system process ID. The process is copied in memory from the parent and a new process structure is assigned by the kernel. The environment **shares** resource limits, umask, controlling terminal, current working directory, root directory, signal masks and other process resources between parent and child. A parent process **does not share** the following with's children: A parent must wait on a child, `fork != 0` for parent (`-1` for error), `fork == 0` for child, Execute asynchrously. **Advantages** (compared to threads): Development is much easier. Fork based code a more maintainable. Forking is much safer and more secure because each forked process runs in its own virtual address space. Fork are more portable than threads. Forking is faster than threading on single cpu as there are no locking over-heads or context switching. **Disadvantages** (compared to threads): In fork, longer startup and stopping time. Inter-process communication is really costly. When the parent exits before the forked child, you will get a ghost process. In-sufficient storage space could lead the fork system to fail.

```
pid_t pid = fork();
switch (pid) {
case -1: perror("fork failed\n"); exit(1);
case 0: /* This is the child" */ break;
default: /* "This is the parent" */ break; }
```

The `exec()` family of functions will initiate a program from within a program. They are also various front-end functions to `execve()`. The functions return an integer error code. The function call `execl()` initiates a new program in the same environment in which it is operating. An executable (with fully qualified path. i.e. `/bin/ls`) and arguments are passed to the function. Note that `arg0` is the command/file name to execute. Use `exec` to make a child process execute a new program after it has been forked. It usually does not return, with `-1` on failure. It will fail when: too big, ACCESS, get into a loop, the name is too long, or the executable does not exist. The new process keeps the set of blocked signals, pending signals, timers, and any open file descriptors. Doing an `exec` does not change the relationship between a parent and child process. Caught signals return default values.

```
execl("/bin/ls","/bin/ls","-r","-t",(char*)0);
```

`wait()`: Blocks calling process *until the child process terminates.* If child process has already terminated, the wait() call returns immediately. if the calling process has multiple child processes, the function returns when one returns. `waitpid()`: Options available to block calling process for a particular child process not the first one.

Shell scripting

`m >n`: is file descriptor, n is file name, `m >& n`: both file descriptors, `1 >& 2`: redirect standard out to standard error, `& >`: redirect both things to a standard file, `command >>file` = append to file, `command <file`: contents to file to standard in, `grep searchword <filename`, &&: if left fails, stop, else return 0 and do right, `||`: if left succeeds don't do right, `grep [options] regex files`: returns files matching regex, `-i` = ignore case, `-v`: print unmatched, `-l` print filenames and not lines, `ls -a(all) -l(longform)` , (`ps -e`: all process ID's on machine) (`-u` all IDs on user), (`sh` = invoke another shell).

```bash
#!/bin/bash
echo My name is $0
echo My process number is $$
echo I have $# arguments
echo My arguments separately are $*
echo My arguments together are "$@"
echo My 5th argument is "'$5'"
echo The return value of last command $?
usage() {
 echo "Usage: $0 filename"
 exit 1 }
is_file_exits(){
 local f="$1"
 [[ -f "$f" ]] && return 0 || return 1 }
# invoke  usage
# call usage() function if filename not supplied
[[ $# -eq 0 ]] && usage
# Invoke is_file_exits
if ( is_file_exits "$1" )
then
 echo "File found"
else
 echo "File not found"
fi
```

Shared memory

Possible for multiple processes to share segments of memory, not full memory. Beneficial for better speed. Access by multiple processes. Still need mutexes and semaphore for synchronizations. Lifetime is independent of any process. All shared memory segments exist in a single list. Identified by a key, generated by the metainfo of files and directories. Processes attach and detach to shared memory segments. Used in data structures for shared memory, `shmat` maps shared memory. `shmdt` unmaps shared memory. When using data structures, use shared offsets such as `(struct something) shmat(...)` or use semaphores where second argument isn't zero. **Advantages**: Potential advantages include efficiency (elimination of unnecessary data copying) and reduced complexity (single-step updates rather than the read, modify buffer, write cycle). **Disadvantages**: Synchronizing access is tough, security (accidental overwrite, malicious overwrite).

```c
key_t key; /* key to be passed to shmget() */
int shmflg; /* shmflg to be passed to shmget() */
int shmid; /* return value from shmget() */
int size; /* size to be passed to shmget() */
char *shm; /* data to be passed */

if ((shmid=shmget (key, size, shmflg)) == -1){}
if ((shm=shmat(shmid, NULL, 0))==(char *) -1){}
if (shmdt(shmid) == -1){}
```

Pointers to functions

```c
main() {
    void (*fp)(int); fp = func;
    (*fp)(1); fp(2);
    exit(EXIT_SUCCESS); }
void func(int arg){ printf("%d\n", arg); }
```

Function pointers provide a way of passing around instructions for how to do something. You can write flexible functions and libraries that allow the programmer to choose behavior by passing function pointers as arguments.

Condition variables

Condition variables provide yet another way for threads to synchronize. While **mutexes** implement synchronization by *controlling thread access to data*, **condition variables** *allow threads to synchronize based upon the actual value of data.* Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met. This can be very resource consuming since the thread would be continuously busy in this activity. *A condition variable is a way to achieve the same goal without polling.* A condition variable is *always used in conjunction with a mutex lock.* Used for waiting. Contains 3 basic operations: wait, signal, broadcast. Wait atomically does unlock and block. Signal atomically does re-locks a thread has it start again. A **mutex** only allows you to wait until the lock is available; a *condition variable allows you to wait until some application-defined condition has changed.*

```c
pthread_mutex_t count_mutex=
    PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condition_var=
    PTHREAD_COND_INITIALIZER;
int count = 0;
#define COUNT_DONE 10,COUNT_HALT1 3,COUNT_HALT2 6
main(){
  pthread_t thread1,thread2;
  pthread_create(&thread1,NULL,&count1,NULL);
  pthread_create(&thread2,NULL,&count2,NULL);
  pthread_join(thread1,NULL);
  pthread_join(thread2,NULL);
  exit(0);}
void *count1() {
  for(;;) {pthread_mutex_lock(&count_mutex);
  pthread_cond_wait(&condition_var,&count_mutex);
  count++;
  pthread_mutex_unlock(&count_mutex);
  if(count>=COUNT_DONE)return(NULL);}}
void *count2() {
  for(;;) { pthread_mutex_lock(&count_mutex);
  if(count<COUNT_HALT1||count>COUNT_HALT2){
    pthread_cond_signal( &condition_var ); }
  else { count++; }
  pthread_mutex_unlock( &count_mutex );
  if(count >= COUNT_DONE) return(NULL); } }
```

Deadlock

Four conditions for deadlock: **mutual exclusion** (a resource must be held in a non-sharable mode.), **circular wait** (A process must be waiting for a resource which is being held by another process, which is waiting for the first process to release the resource.), **no pre-emption** (The operating system must not de-allocate resources once they have been allocated.), **hold and wait** (A process is currently holding at least one resource and requesting additional resources which are being held by other processes.).

Semaphores

A *semaphore* is a special type of variable that can be incremented or decremented, but *crucial access to the variable is guaranteed to be atomic*, even in a multi-threaded program. If two or more threads in a program attempt to change the value of a semaphore, the system guarantees that all the operations will in fact *take place in sequence.* Contains a counter,not an owner. The counter is anything that is non-negative and can be posted by anything. Wait has two checks: if counter is not zero, decrement and return immediately, but if it is zero then suspend the calling thread and put it in the semaphore queue. **Advantages**: permit more than one thread to access the critical section, semaphores are machine independent. **Disadvantages**: Modularity is lost.

```c
void handler(void *ptr);sem_t mutex;int counter;
int main() {
 int i[2];i[0]=0;i[1]=1;
 pthread_t a;pthread_t b;sem_init(&mutex, 0, 1);
 pthread_create(&a,NULL,(void*)&h,(void*)&i[0]);
 pthread_create(&b,NULL,(void*)&h,(void*)&i[1]);
 pthread_join(a, NULL); pthread_join(b, NULL);
 sem_destroy(&mutex); exit(0); }
void h ( void *ptr ) {
 int x;x = *((int *) ptr);sem_wait(&mutex);
 counter++; sem_post(&mutex);
 pthread_exit(0); }

sem_init(sem_t *sem,int pshared,unsigned int v);
sem_wait(sem_t *sem); int sem_post(sem_t *sem);
sem_getvalue(sem_t *sem, int *valp);
sem_destroy(sem_t *sem);
```

Libraries

Static: Precompiled with all functions. **Advantages**: Simple, reliable. **Disadvantages**: Large, poor security. **Dynamic**: Loaded on the fly. **Advantages**: Typically smaller, good for security. **Disadvantages**: Hard to test and implement.

```
cc -Wall -c ctest1.c ctest2.c
ar -cvq libctest.a ctest1.o ctest2.o
ar -t libctest.a
cc -o prog prog.c libctest.a
```

Pointers

```c
/* wrong */ void swap(int x, int y) {
int temp; temp = x; x = y; y = temp;}
/* right */void swap(int*px, int*py){
int temp;temp=*px;*px=*py; *py=temp;}
```