IT628: Systems Programming

SIGALRM, race condition

alarm system call

- **unsigned int alarm(unsigned int seconds);**

- **Sets a timer in seconds**
- **When time is up, SIGALRM is sent to the process. Default action is to terminate the process**
- **Only 1 alarm per process**
- **Returns**
  - 0 if no previous alarm set, or if previous alarm had less than 1 second left
  - the number of seconds left before previous alarm
- **alarm(0) cancels any pending alarms**

pause system call

- int pause(void);

  - Suspends a process until it receives a signal
  - Returns -1
  - Can use alarm and pause to put a process to sleep for a specified amount of time

signal2.c

- This program prints a message every second:

```
/* Print a message, then request another SIGALRM. */
void handle_sigalrm(int sig) {
   printf("Hello!\n");
   alarm(1); /* Request another SIGALRM in 1 second. */
}
/* User typed Ctrl-C. Taunt them. */
void handle_sigint(int sig) {
    printf("Ha ha, can't kill me!\n");
}
int main() {
    signal(SIGINT, handle_sigint);
    signal(SIGALRM, handle_sigalrm);
    alarm(1); /* Request a SIGALRM in 1 second. */

    while (1) pause(); /* Gentle infinite loop. */
    return 0;
}
```

Race conditions: An Example

```
int i = 0;

void handler(int sig) {
   i++;
}

int main()  {
   int temp;

   signal(SIGALRM, handler);
   alarm(1);

   temp = i;
   temp = temp + 1;
   i = temp;

   pause();
   printf("%d\n", i);
}
```

What is the output?

Depends on when the signal handler is called

If the alarm is caught during the execution of these three lines then the program outputs 1, instead of 2!

Race conditions and Signals

- **Suppose both the main thread of execution and a signal handler modify shared state**
- **If the main thread is modifying this state and the signal handler is called during this operation, the signal handler sees *inconsistent state***

Race conditions: An Example

```
int i = 0;

void handler(int sig) {
   i++;
}

int main()  {
   int temp;

   signal(SIGALRM, handler);
   alarm(1);

   temp = i;
   temp = temp + 1;
   i = temp;

   pause();
   printf("%d\n", i);
}
```

How do we make this program deterministic (i.e. make it output 2 on every execution)?

Block the alarm during the critical section

Critical section

Function for Blocking Signals

```
int sigprocmask(int iHow,
                const sigset_t *psSet,
                sigset_t *psOldSet);
```

`psSet`: **Pointer to a signal set**

`psOldSet`: **(Irrelevant for our purposes)**

`iHow`: **How to modify the signal mask**

- `SIG_BLOCK`: **Add `psSet` to the current mask**
- `SIG_UNBLOCK`: **Remove `psSet` from the current mask**
- `SIG_SETMASK`: **Install `psSet` as the signal mask**

Functions for constructing signal sets

- `sigemptyset()`, `sigaddset()`, …

Race condition: Fix using sigprocmask

```
int i = 0;
void handler(int sig) { i++; }
int main()  {
   int temp;
   sigset_t sSet;

   signal(SIGALRM, handler);
   alarm(1);

   sigemptyset(&sSet);
   sigaddset(&sSet, SIGALRM);
   sigprocmask(SIG_BLOCK, &sSet, NULL);

   temp = i;
   temp = temp + 1;
   i = temp;

   sigprocmask(SIG_UNBLOCK, &sSet, NULL);

   pause();
   printf("%d\n", i);
}
```

How do we make this program output 2 on every execution)?

Block SIGALRM signal

Critical section

Unblock SIGALRM signal

Exercise

- Write a program that continuously prompts the user for a string, and then prints the string in uppercase. Have your program catch the signal SIGINT and print a message instructing the user to instead use CONTROL-D to exit the program. In addition, have your program re-prompt the user for input after 10, 20, and 30 seconds of inactivity. After 40 seconds of inactivity, your program should exit. (Hint: use alarm() and a signal handler for SIGALRM).

Signal disposition after forking

- A newly forked child inherits its parent's signal handlers!
  - because fork() creates a copy of the parent, addresses of signal handlers are still valid for the child
- but does not inherit any pending signals or alarms

Demos: sigfork1.c, sigfork2.c

sigfork1.c

```
/* Handle SIGALRM */
void handler(int sig)
{
  if (cpid == 0)
     printf("running child
                  handler\n");
  else
     printf("running parent
                  handler\n");
}
```

```
pid_t cpid;

int main() {
   signal(SIGALRM, handler);
   if ((cpid = fork()) == 0) {
     printf("I'm the child\n");
     alarm(2);
     pause();
     printf("child pause ret\n");
   }
   else {
     printf("I'm the parent\n");
     alarm(4);
     pause();
     printf("parent pause ret\n");
     wait(NULL);
   }
}
```

sigfork2.c

```
/* Handle SIGALRM */
void handler(int sig)
{
  if (cpid == 0)
     printf(“running child
                  handler\n”);
  else
     printf(“running parent
                  handler\n”);
}
```

```
pid_t cpid;

int main() {
   int retval;
   signal(SIGALRM, handler);
   alarm(2);
   if ((cpid = fork()) == 0) {
     printf(“I’m the child\n”);
     retval = sleep(10);
     printf(“child retval=%d\n”, retval);
   }
   else {
     printf(“I’m the parent\n”);
     retval = sleep(10);
     printf(“parent retval=%d\n”);
     wait(NULL);
   }
}
```

Signal disposition after execing

- Processes lose their signal handlers when execing a new program
  - the address of the signal handler no longer exists in the new program
  - set back to the default handler
- But
  - previously ignored signals remain ignored
  - new process image inherits the time left to an alarm signal from the old process image

Demos: sigexec1.c, sigexec2.c

sigexec1.c

- **do_nothing is a program with an infinite loop displaying "Do nothing program" every second**

```
void handler(int sig) {
   printf("Can't be ctr-\\ed\n");
}

int main()
{
   signal(SIGINT, SIG_IGN);
   signal(SIGQUIT, handler);
   pause();
   execl("./do_nothing", "do_nothing", NULL);
}
```

sigexec2.c

- **do_nothing is a program with an infinite loop displaying “Do nothing program” every second**

```
int main()
{
   if (fork() == 0) {
      alarm(5);
      printf(“I am the child\n”);
      sleep(2);
      execl(“./do_nothing”, “do_nothing”, NULL);
   }
   else {
      alarm(4);
      while (1) {
         printf(“I am the parent\n”);
         sleep(1);
      }
   }
}
```

Output

The do_nothing program terminates because of the alarm(5) call. It terminates after the time left when exec is called (roughly about 5-2=3 seconds)

```
I am the parent
I am the child
I am the parent
Do nothing program
I am the parent
Do nothing program
I am the parent
Alarm clock
Do nothing program
```