1 Exception Control Flow

Understanding ECF can help us

- Understand important concepts: IO, process, virtual memory, etc.
- Understand how applications interact with the OS: writing data to disk, reading data from network, creating / terminating a process all involve a form of ECF known as **system call (trap)**.
- Write interesting new programs such as Unix shells and web servers, using ECF mechanisms provided by the OS.
- Understand concurrency.
- Understand how software exceptions work in C++, JAVA, etc.

1.1 Exception

1.1.1 Definition

Exception An abrupt change in the control flow in response to some change the processor's state.

Event Change in a processor's state (encoded in various bits and signals).

- Can be related to the execution of the current instruction: virtual memory page fault, arithmetic overflow, division by 0, etc.
- Or unrelated: system timer goes off, I/O request completion.

When the processor detects an event, it makes an indirect procedure call (the exception), through a jump table called the **exception table**, to an OS subroutine (the exception handler). When the exception handler finishes processing, it either returns control to the current instruction or the next instruction, or aborts the program.

Exception number An unsigned integer assigned to each type of exception.

- By processor designer: division by 0, page fault, illegal memory access, break point, arithmetic overflow.
- By OS kernel designer: system call, signals from external IO devices.

Exception table Allocated at system boot time. Entry k contains the address of the handler of exception k. Its starting address is stored in a special CPU register called the **exception table base register**.

Exception is akin to procedure call except for some important differences:

- As with procedure call, the return address is pushed on the stack. But the return address can be the current or the next instruction.
- The processor also pushes some additional processor state on the stack that will be necessary when the handler returns and the interrupted program continues, e.g. EFLAGS.

- When control is transferred to the kernel, these items are pushed onto the kernel stack instead of the user stack.
- Exception handlers run in kernel mode, thus have complete access to all system resources.

1.1.2 Classes

Async Not caused by any instruction.

Sync Caused by the current instruction (the faulting instruction).

Table 1: Classes of exceptions

Class	Cause	Async/Sync	Return address
Interrupt	Signal from IO devices	Async	The next instruction
Trap	Intentional exception	Sync	The next instruction
Fault	Potentially recoverable error	Sync	The current instruction
Abort	Non-recoverable error	Sync	Never returns

Table 2: Example of x86-64 exceptions

Number	Description	Class	Outcome
0	Division by 0 (Floating exception)	Fault	No attempt to recover
13	General protection fault (Segmentation fault)	Fault	No attempt to recover
14	Page fault	Fault	Maps disk page to memory page.
18	Machine check (Fatal hardware error)	Abort	
32-255	OS defined exceptions	Interrupt or trap	

syscall C function & x86-64 instruction to make system calls. For the instruction:

- System call number: `%rax`
- Arguments: `%rdi, %rsi, %rdx, %r10, %r8, %r9`
- Return value: `%rax`. -4095 $\sim$ -1 indicates error, corresponding to negative `errno`.
- Registers destroyed: `%rcx, %r11`

"Hello world" written using system-level function:

```
int main() {
  write(1, "Hello, world\n", 13); //1: stdout. 13: number of bytes to write.
  _exit(0);
}
```

Table 3: Example of x86-64 system calls

N.O.	Name	Description	N.O.	Name	Description
0	`read`	Read file	33	`pause`	Suspend process until signal arrives
1	`write`	Write file	37	`alarm`	Schedule delivery of alarm signal
2	`open`	Open file	39	`getpid`	Get process ID
3	`close`	Close file	57	`fork`	Create process
4	`stat`	Get info about file	59	`execve`	Execute a program
9	`mmap`	Map memory page to file	60	`_exit`	Terminate process
12	`brk`	Reset the top of the heap	61	`wait4`	Wait for a process to terminate
32	`dup2`	Copy file descriptor	62	`kill`	Send signal to a process

1.2 Processes

Process An instance of a program in execution. Each program runs in the context of a process. A process provides to applications 2 key abstractions:

- An independent **logical control flow** that provides the illusion that this program has exclusive use of the processor.
- A **private address space** that provides the illusion that this program has exclusive use of the memory system.

Context The state that the program needs to run correctly, including the program's code and data stored in the memory, its stack, general registers' contents, PC, environment variables, and the set of open file descriptors.

Concurrent flow A logical flow whose execution overlaps with another logical flow.

Concurrency The general phenomenon of multiple flows executing concurrently.

Multi-tasking(time slicing) The notion of a process taking turns with other processes. Processes get **preempted** by each other.

Time slice Each time period that a process executes a portion of its flow.

Parallel flows A proper subset of concurrent flows: two flows running concurrently on different processor cores or computers.

Private address space Private in the sense that the byte related to an address cannot be read-/written by other processes.

User/Kernel mode A mechanism provided by the processor to restrict the instructions that an application can execute and the portions of the address space that it can access.

- Controlled by the **mode bit** in a control register on the processor.
- A process in user mode cannot execute **privileged instructions**, e.g. halt the processor, change the mode bit, start an IO operation.
- User programs can only access kernel code and data via system call.

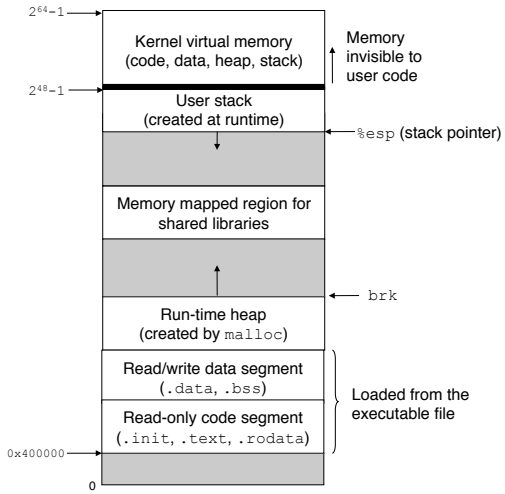

Figure 1: Structure of private address space

- Exception handler switches the processor to kernel mode, and switches back when it returns.
- `/proc` filesystem allows user mode processes to access the contents of kernel data structures.

Context switches A higher-level ECF used by the OS kernel to implement multi-tasking. It saves the context of the current process; restores the saved context of some previously preempted process; passes control to this newly restored process. Such decision is called **scheduling**, which is handled by some kernel code known as the **scheduler**.

1.3 System call error handling

Error-reporting function:

```
void unix_error(char *msg) {
  fprintf(stderr, "%s: %s\n", msg, strerror(errno));//strerror in string.h
  exit(0);
}
```

Error-handling wrapper (using `fork`) as an example:

```
pid_t Fork(void) {
  pit_t pid;
  if((pid = fork()) < 0)
    unix_error("Fork error");
  return pid;
}
```

1.4 Process control in Unix

1.4.1 Obtain process ID

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void); //get pid of parent process.
```

1.4.2 Create/Terminate processes

From the perspective of a programmer, a process is always in one of the 3 states:

Running Either executing on CPU or waiting to be scheduled.

Stopped Suspended and won't be scheduled.

- Happens after receiving signals SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU.
- Remains stopped until receiving signal SIGCONT, after which the process becomes running again.

Terminated Permanently stopped. Happens after:

- Receiving signals whose default action is to terminate the process.
- Returning from the main routine.
- Calling the `exit` function. `status` is the **exit status**, which can also be set by returning an integer value from the main routine.

  ```
  #include <stdlib.h>
  void exit(int status);
  ```

A **parent process** creates a **child process** with the `fork` function:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

- The child gets an identical yet separate copy of the parent's user-level virtual memory.
- The child also gets identical copies of the parent's open file descriptors.
- The parent and the child have different PIDs.
- In the parent, `fork` returns the pid of the child; in this child, it returns 0.
- The parent and the child are executed concurrently.

1.4.3 Reaping child processes

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *statusp, int options);
```

1. Default behaviour (`options=0`):
   - Elements of **wait set**:
     - `pid>0`: the single process whose ID is `pid`.
     - `pid=-1`: all child processes of the parent process.
   - Suspend the calling process until one of the child processes in its wait set terminates.
   - If a process in the wait set already terminated, return immediately.
   - The pid of the child process is returned. It is reaped.
2. Other options:

   WNOHANG Return 0 immediately if no child process has terminated.

   WUNTRACED Terminated → terminated or stopped.

   WCONTINUED Suspend the calling process, until a running process in the wait set terminates or a stopped process in the wait set resumes after receiving signal SIGCONT.

   WNOHANG|WUNTRACED The options can be combined by |.
3. If `statusp` is not NULL, then the status of the reaped child process is saved in the value `status` pointed to by `statusp`. It can be explained with the following macros defined in `wait.h`:
   - WIFEXITED(`status`): true if the child process terminated normally via `exit` or `return`.
   - WEXITSTATUS(`status`): returns the exit status of a normally terminated child. Only defined if WIFEXITED() is true.
   - WIFSIGNALED(`status`): true if the child process terminated because of an uncaught signal.
   - WTERMSIG(`status`): returns the number of the signal that terminated the child process. Only defined if WIFSIGNALED() is true.
   - WIFSTOPPED(`status`): true if the child process is stopped.
   - WSTOPSIG(`status`): the number of the signal that caused the child to stop. Only defined if WIFSTOPPED() is true.
   - WIFCONTINUED(`status`): true if the child process restarted after receiving signal SIGCONT.
4. If the calling process has no children, `waitpid` returns -1 and sets `errno` to ECHILD. If `waitpid` is interrupted by a signal, it returns -1 and sets `errno` to EINTR.
5. Simplified version:

```
pid_t wait(int *statusp); //equivalent to waitpid(-1, statusp, 0)
```

1.4.4 Putting process to sleep

```
#include <unistd.h>
unsigned int sleep(unsigned int secs);
int pause(void); //always returns -1
```

`sleep` returns 0 or the number of seconds left to sleep. The latter is possible because it can be interrupted by a signal. `pause` puts the calling process to sleep until it receives a signal.

1.4.5 Load & run programs

```
#include <unistd.h>
int execve(const char* filename, const char *argv[], const char *envp[]);
//related main function:
int main(int argc, char **argv, char **envp);
```

- Does not return to the calling process, unless an error happens (e.g cannot file the file).
- `argv` points to a `null`-ending array of pointers. Each pointer points to an argument string. `argv[0]` is the name of the executable.
- `envp` points to a `null`-ending array of pointers. Each pointer points to an environment variable string, e.g. `name=value`.

To manipulate the environment array:

```
#include <stdlib.h>
char *getenv(const char* name); //NULL if name does not exist
int setenv(const char* name, const char* value, int overwrite);//0 for success, -1
    for failure.
void unsetenv(const char*);
```

1.5 Signals

Signal A small message that informs a process of an event of some type that has happened in the system.

Pending signal A sent but not received signal. There is at most one pending signal of a particular type. If there exists already a pending signal of a type, following signals of the same type are simply discarded. The kernel maintains the pending bit vector (signal mask) for each process.

Blocked signal Can be sent but won't be received. The kernel maintains the blocked bit vector for each process.

Process group Each process belongs to one process group. By default a child process belongs to the same process group as its parent.

```
#include <unistd.h>
//Return process group ID of the current process
pid_t getpgrp(void);
```

```c
/* Change process group id of the process pid to pgid.
   If pid = 0, act on the current process.
   If pgid = 0, use the PID of the process specified by pid as the group ID.
   return 0 for success, -1 for error. */
int setpgid(pid_t pid, pid_t pgid);
```

1.5.1 Send signals

1. Use `/bin/kill` program. e.g. `/bin/kill -9 15123` sends 9 (SIGKILL) to process 15123. A negative PID causes the signal to be sent to all processes in the process group with the ID.
2. Send from keyboard.
   - A job represents processes created to evaluate a command line in Unix shell. Unix shell creates a separate process group for each job.
   - At any time, there can be at most 1 foreground job and 0 or more background jobs.
   - Ctrl + C sends SIGINT (default behaviour: terminate) to each process in the foreground process group.
   - Ctrl + Z sends SIGTSTP (default behaviour: suspend) to each process in the foreground process group.
3. Use `kill` function to send signals to other processes (including itself).

```c
#include <sys/types.h>
#include <signal.h>
/* pid > 0: send sig to pid.
   pid = 0: send sig to all processes in the same process group as the
       calling process.
   pid < 0: send sig to all processes in the process group |pid|
   return 0 for success, -1 for error. */
int kill(pid_t pid, int sig);
```

4. Use `alarm` function to send SIGALRM signal to the process itself. Note that the default behaviour of the signal is to terminate.

```c
#include <unistd.h>
/* Send SIGALRM to the calling process after secs seconds. secs = 0: no new
    alarm is scheduled.
Cancels any pending alarm.
Returns the number of remaining seconds of the pending alarm, or 0 if there
    isn't any pending alarm.*/
unsigned int alarm(unsigned int secs);
```

1.5.2 Receive signals

Predefined default behaviour after receiving a signal can be:

- The process terminates.
- The process terminates and dumps core.

- The process suspends until restarted by SIGCONT.
- The process ignores the signal.

The default behaviour of SIGSTOP and SIGKILL cannot be modified. The default behaviour of other signals can be modified with `signal` function.

```
#include <signal.h>
typedef void (*sighandler_t)(int);
/*
handler = SIG_IGN: ignore signal signum.
handler = SIG_DFL: restore default behaviour of signal signum.
Otherwise: install new signal handler for signum.
Return previous handler for success. SIG_ERR for error.
*/
sighandler_t signal(int signum, sighandler_t handler);
```

1.5.3 Block/Unblock signals

Implicit blocking mechanism The kernel blocks any pending signals of the type currently being processed by a handler.

Explicit blocking mechanism Use `sigprocmask` and its helper functions.

```
#include <signal>
/*
how = SIG_BLOCK: blocked = blocked | set.
how = SIG_UNBLOCK: blocked = blocked & ~set.
how = SETMASK: blocked = set.
return 0 for success, -1 for error. Old value of blocked is kept in oldset (if
    oldset is not NULL).*/
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);

// set manipulation functions. Return 0 for success, -1 for error.
int sigemptyset(sigset_t *set); //Initialize as empty set
int sigfillset(sigset_t *set); //Add all signals to the set
int sigaddset(sigset_t *set, int signum); // Add signum
int sigdelset(sigset_t *set, int signum); //Delete signum

//Check if signum is member of set. Return 1 for yes, 0 for no and -1 for error.
int sigismember(const sigset_t *set, int signum);
```

1.5.4 Writing signal handlers

1. Handlers run concurrently with the main program and share the same global variables, and thus can interfere with the main program and other handlers.
2. The rules for how / when signals are received are often counterintuitive.
3. Different systems may have different signal handling semantics.

Safe signal handling Some conservative guidelines:

1. Keep handlers as simple as possible, e.g. set a global flag and return immediately, and the main program checks the flag periodically for further processing associated with the receipt of the signal.
2. Call only async-signal-safe functions in handlers. Such functions are either reentrant or unable to be interrupted by a signal handler.
3. Save `errno` on entry to the handler and restore it when the handler returns to avoid interference with other parts of the program that rely on `errno`.
4. Block all signals when accessing shared global data structures. Accessing a data structure d from the main program typically requires a sequence of instructions. If the sequence is interrupted by a handler that also accesses d, d may end up in an inconsistent state.
5. Declare global variables as `volatile` to forbid the compiler to cache them.
6. Declare flags with `sig_atomic_t` to guarantee atomic read/write.

Correct signal handling Pending signals are not queued.

```
//SIGCHLD handlers

//Buggy!
void handler1(int sig) {
  int olderrno = errno;
  if(waitpid(-1, NULL, 0) < 0) //1 reaped. 1 pended. Other possibly discarded!
    sio_error("waitpid error");
  Sleep(1);
  errno = olderrno;
}

//Correct
void handler2(int sig) {
  int olderrno = errno;
  while(waitpid(-1, NULL, 0) > 0); //Reaps as many children as possible.
  if(errno != ECHILD)
    Sio_error("waitpid error");
  Sleep(1);
  errno = olderrno;
}
```

Portable signal handling `sigaction` function and `Signal` wrapper to resolve historical incompatibility. Not something interesting.

1.5.5 Synchronize flows to avoid concurrency bugs

How to program concurrent flows that read and write the same storage locations? Synchronize concurrent flows to allow the largest set of instruction interleavings that produce the correct result.

The following program describes the typical structure of a Unix shell.

```
void handler(int sig) {
  int olderrno = errno; /* Safe signal handling 3 */
  pid_t pid;
  sigset_t mask_all, prev_all;
  Sigfillset(&mask_all);
  while((pid = waitpid(-1, NULL, 0)) > 0) { /*Safe signal handling 4*/
    Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
    deletejob(pid);
    Sigprocmask(SIG_SETMASK, &prev_all, NULL);
  }
  if(errno != ECHILD)
    Sio_error("waitpid error");
  errno = olderrno;
}
int main(int argc, char** argv) {
  pid_t pid;
  sigset_t mask_all, prev_all;

  Sigfillset(&mask_all);
  Signal(SIGCHLD, handler);
  initjobs(); /*Initialize job list*/

  while(1) {
    if((pid = Fork()) == 0) { /*child*/
      Execve("/bin/date", argv, NULL);
    }
    Sigprocmask(SIG_BLOCK, &mask_all, &prev_all); /*parent. Safe signal handling 4*/
    addjob(pid);
    Sigprocmask(SIG_SETMASK, &prev_all, NULL);
  }
  exit(0);
}
```

Unfortunately it will potentially cause race condition. If the child is scheduled to run after the parent calls `Fork`, and completes before the parent is scheduled to run again, a SIGCHLD will be sent to the parent, so that `deletejob` will be called before `addjob`.

The solution is to block SIGCHLD before calling `Fork`. Note that the child inherits the block set from it parent, so SIGCHLD has to be unblocked before calling `Execve`.

```
int main(int argc, char** argv) {
  pid_t pid;
  sigset_t mask_all, mask_one, prev;

  Sigfillset(&mask_all);
  Sigemptyset(&mask_one);
  Sigaddset(&mask_one, SIGCHLD);
  Signal(SIGCHLD, handler);
  initjobs();

  while(1) {
    Sigprocmask(SIG_BLOCK, &mask_one, &prev);
    if((pid = Fork()) == 0) {
      Sigprocmask(SIG_SETMASK, &prev, NULL); //Unblock SIGCHLD
      Execve("/bin/date", argv, NULL);
    }
    Sigprocmask(SIG_BLOCK, &mask_all, NULL);
    addjob(pid);
```

```
    Sigprocmask(SIG_SETMASK, &prev, NULL);
  }
}
```

1.5.6 Explicitly waiting for signals

Sometimes the main program has to explicitly wait for a handler to run. For example, after a foreground job is created in Linux shell, another command can be processed only after the job completes with a SIGCHLD signal.

```
volatile sig_atomic_t pid;
void sigchld_handler(int s) {
  int olderrno = errno;
  pid = waitpid(-1, NULL, 0);
  errno = olderrno;
}
void sigint_handler(int s) {}

int main(int argc, char** argv) {
  sigset_t mask, prev;
  Signal(SIGCHLD, sigchld_handler);
  Signal(SIGINT, sigint_handler);
  Sigemptyset(&mask);
  Sigaddset(&mask, SIGCHLD);

  while(1) {
    Sigprocmask(SIG_BLOCK, &mask, &prev); //to avoid race condition. SIGCHLD being
        received before setting pid to 0 results in infinite loop.
    if(Fork() == 0) /* child */
      exit(0);

    /* parent */
    pid = 0;
    Sigprocmask(SIG_SETMASK, &prev, NULL);

    while(!pid);

    //work after receiving SIGCHLD
    printf(".");
  }
  exit(0);
}
```

The code is correct but the while loop is wasteful. Changing it to

```
while(!pid) //we still need a loop because of SIGINT
  pause();
```

causes race condition: SIGCHLD may arrive after the test in `while` and before `pause`, which causes infinite loop. Changing `pause()` to `sleep(1)` is correct but too slow. The appropriate solution is to use `sigsuspend`.

```
#include <signal.h>
int sigsuspend(const sigset_t *mask);
```

It's equivalent to an atomic version of:

```
sigprocmask(SIG_SETMASK, &mask, &prev);
pause();
sigprocmask(SIG_SETMASK, &prev, NULL);
```

`while(!pid);` should be changed to

```
while(!pid)
  sigsuspend(&prev);
//SIGCHLD is still blocked here. Unblock it.
Sigprocmask(SIG_SETMASK, &prev, NULL);
```

1.6 Nonlocal jump

```
#include <setjmp.h>

int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int retval);

//version of setjmp and longjmp that can be used by handlers
int sigsetjmp(sigjmp_buf env, int savesigs);
int siglongjmp(sigjmp_buf env, int retval);
```

`setjmp` saves the current calling environment in the `env` buffer, including PC, stack pointer and general-purpose registers. It returns 0 here, but its return value should not be assigned to a variable. Yet it can safely be used in a `switch` or conditional statement.

`longjmp` restores the calling environment from `env` and triggers a return from the most recent call of `setjmp` that initialized `env`. The `setjmp` returns with non-zero value `retval`.

The exception mechanisms in JAVA and C++ are higher-level, more structured versions of `setjmp` and `longjmp`. A `catch` clause inside a `try` statement is akin to a `setjmp` function, and a `throw` statement is akin to a `longjmp` function.