Chapter 6

Processes

Read…

- **BLP:** Chapter 11
- **Background on Processes in Operating Systems**
  - Silberschatz: 4.1–4.3
  - Tanenbaum: 2.1
- (**USP: Chapter 3 — Processes in UNIX**)

Sequential Processes on a Timesharing System

A process is a **program in execution**.

A process **needs resources** to accomplish its task: CPU time, memory, files, and I/O devices.

The **operating system** is responsible for the following activities in connection with process management:

- Process **creation** and **deletion**
- Process **suspension** and **resumption**
- Provision of **mechanisms** for:
  - process **synchronization,** *e.g. for avoiding conflicts*
  - inter-process communication, *e.g. for letting servers reload their configuration files*

Process Management Overview

- A **process** is a program in execution
- The CPU is **shared** among the processes **ready to run**
  - Only one process can run at a time
  - The CPU is rapidly switched between processes — **"context switch"**
- Processes communicate with the OS using **system calls**
- A process **can be interrupted at any time** by a device interrupt or a system call
- A process is represented by a data structure called a **process control block (PCB)** or a **process descriptor**

Context Switching

A **context switch** saves the state of the current running process and then loads the state of the next process to be executed.

It needs to save enough information about the current running process so that it can be resumed later *as if nothing had happened*. This determines important components of the PCB.

A context switch is initiated by an interrupt:

– **software interrupt** (system call)

– **device interrupt**

– **timer interrupt** (quantum expired)

"Calling BIOS Routines"

From SE3F, you know how to "*call a BIOS routine*" to perform actions such as output of a character to screen:

– Prepare certain registers to contain certain values

– Call "`INT` *XY*`h`"

How does the CPU execute the `INT` instruction?

CPU Interrupt Handling — "`INT` *XY*`h`"

- The CPU switches into **privileged mode**
- The lowest ranges of main memory contain the **interrupt vector**
- *XY*`h` is used as index into the interrupt vector to retrieve an *address*
- That address is transferred into the program counter (PC)

So `INT` is a complicated kind of indirect jump …

What happens then?

- That address will have pointed to the start of some **interrupt handler** (routine) which is then executed.
- These interrupt handlers are part of the kernel.
- The kernel is responsible to change back to user mode before resuming execution of a user process.

Context Switch Steps

(Assuming a user process is interrupted)

– CPU senses interrupt

– CPU starts the interrupt handler **in privileged mode**

– *(Interrupt handler may disable all interrupts)*

– Interrupt handler stores state of interrupted process

– Interrupt handler executes (kernel) code for the interrupt

– Interrupt handler ends by calling the CPU scheduler

– CPU scheduler loads the state of the process it selects

– *(Interrupt handler would re-enable interrupts)*

– CPU switches back to **user mode** and executes selected process

PCB Components

- Process identity — UNIX: *pid_t getpid*(void)
- User information — UNIX: *uid_t getuid*(), *geteuid*()
- Process state
- Program counter
- CPU register values
- Allocated memory (for code, data, stack, etc.)
- Allocated resources (I/O devices, files, etc.)
- CPU-scheduling information
- Any other needed information about the process

Process States

- **new**: A process is being created
  - changes to **ready** when created
- **ready/runnable**: A process is ready to be dispatched
  - changes to **running** when dispatched
- **running**: The process's program is being executed
  - changes to **blocked** when issuing a waiting syscall (I/O)
  - changes to **ready** when a timer interrupt occurs
  - changes to **terminated** when process terminates
- **blocked**: The process is **waiting** for I/O or a message
  - changes to **ready** when I/O is done or message is received
- **terminated**: The process has terminated (“*zombie*”)

Process Scheduling

- All processes are on the **job queue / process table**
  - Some *ready* processes may be swapped to disk.
  - Processes *in memory* that are ready to execute are also on the **ready queue**
  - Processes waiting for an I/O device are also on its **device queue**
- The **job scheduler** selects processes from the job queue to load into and out of memory
- When the CPU is free, the **CPU scheduler** selects one process from the ready queue to be executed by the CPU
  - A process can run until a timer interrupt or some other interrupt occurs
  - The CPU scheduler is called after the processing of each interrupt

Process Information in UNIX

- POSIX/Solaris: `ps -ef`, Solaris: `/usr/ucb/ps -ucax`

  Linux: `ps ucax`
- **top(1)**
- `/usr/bin/time` and shell-builtin `time`

```
example% time find / -name csh.1 -print
/usr/share/man/man1/csh.1
95.0u 692.0s 1:17:52 16% 0+0k 0+0io 0pf+0w

example% /usr/bin/time find / -name csh.1 -print
/usr/share/man/man1/csh.1
real  1:23:31.5
user     1:33.2
sys     11:28.2
```

Process Information in UNIX — getrusage(2)

```
struct rusage {
  struct timeval  ru_utime;  /* user time used */
  struct timeval  ru_stime;  /* system time used */
  long ru_maxrss;    /* maximum resident set size */
  long ru_idrss;     /* integral resident set size */
  long ru_minflt;    /* page faults not requiring physical I/O */
  long ru_majflt;    /* page faults requiring physical I/O */
  long ru_nswap;     /* swaps */
  long ru_inblock;   /* block input operations */
  long ru_oublock;   /* block output operations */
  long ru_msgsnd;    /* messages sent */
  long ru_msgrcv;    /* messages received */
  long ru_nsignals;  /* signals received */
  long ru_nvcsw;     /* voluntary context switches */
  long ru_nivcsw;    /* involuntary context switches */
};
```

system — Library Function for Program Execution

- int *system*(const char ∗*command*);
- Passes *command* to “`/bin/sh -c`”
- Blocks — i.e., returns after *command* has completed
- But command could go to background ... &
- Haskell: $\mathit{System}.\mathit{system} :: \mathit{String} \rightarrow \mathit{IO}\ \mathit{ExitCode}$

Process Creation — General

- One process (the **parent**) can create another process (its **child**)
- The child’s resources can be
  - allocated by the operating system,
  - obtained from the parent, or
  - shared with the parent.
- The child’s address space can
  - ba a copy of its parent’s address space, or
  - have its own program loaded into it.
- The parent can execute
  - independently of its child. or
  - wait until its child terminates.

exec

The **exec** functions overwrite the calling process’s program:

- the **...p** variants search for the command in $PATH
- the **...e** allow to explicitly pass the **environment**
- **arguments** can be passed as an **array** (**v**ector), or via “***varargs* l**ists”

These functions return **only on error!**

```
int execl (const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char *const envp[]);
int execv (const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *file, char *const argv[], char *const envp[]);
```

Unix-Style Process Creation

Process creation in Unix involves four system calls:

- **fork**: Creates an exact copy of the calling process
  - Returns process id of child to parent
  - Returns **0** to child
- **execve**: Overwrites the calling process's program
- **exit**: Causes the calling process to terminate
  - Returns process id to waiting processes
- **wait**: Causes the calling process to wait until one of its children exits
  - Returns process id of exited child to process

fork

fork: Creates an **exact copy** of the calling process

- only PID and PPID are different
- all data is **copied** (lazily)
- both processes run the **same code**
- both processes are **at the same point** in the execution
- **different return values after fork():**
  - returns PID of child to parent
  - returns **0** to child

USP Example 3.5

```
#include <stdio.h>                              /* simplefork.c */
#include <unistd.h>

int main(void) {
  int x;

  x = 0;
  fork();
  x = 1;
  printf("I am process %ld; x=%d\n", (long)getpid(), x);
  return 0;
}
```

USP Example 3.6

```
#include <stdio.h> <unistd.h> <sys/types.h>  /* abbreviated */
                                               /* twoprocs.c */
int main(void) {
  pid_t childpid;

  childpid = fork();
  if (childpid == –1) {
    perror("Failed to fork");
    return 1;
  }
  if (childpid == 0)                      /* child code */
    printf("I am child %ld\n", (long)getpid());
  else                                   /* parent code */
    printf("I am parent %ld\n", (long)getpid());
  return 0;
}
```

Concurrent Actions

```
#include <stdio.h>                          /* interleaving1.c */
#include <unistd.h>
#include <sys/types.h>
int main(void) {
  pid_t childpid;
  int i,k;
  childpid = fork();
  if (childpid == –1) { perror("Failed to fork"); return 1; }

  k =  (childpid == 0) ? 1 : 10; /* distinguish parent and child */
  for (i=0; i<10; i++) {
    printf("%2d: %2d --- PId = %ld\n", k, k ∗ i, (long)getpid());
    usleep((10 + k) ∗ 20000);
  }
  return 0;
}
```

USP Example 3.7 (modified)

```
#include <stdio.h> <unistd.h> <sys/types.h>  /* abbreviated */
                                             /* badprocessID.c */
int main(void) {
  pid_t childpid, mypid;

  mypid = getpid();
  childpid = fork();
  if (childpid == –1) { perror("Failed to fork"); return 1; }
  if (childpid == 0)                        /* child code */
    printf("I am child %ld, ID = %ld\n",
                (long)getpid(), (long)mypid);
  else                                      /* parent code */
    printf("I am parent %ld, ID = %ld\n",
                (long)getpid(), (long)mypid);
  return 0;
}
```

USP Program 3.1 — A Chain of Processes (modified)

```
#include <stdio.h> <stdlib.h> <unistd.h>  /* abbrev. */
                                          /* simplechain.c */
int main (int argc, char ∗argv[]) {
  pid_t childpid = 0;
  int i, n;
  if (argc ≠ 2)   /* check for no. of cmd-line args */
   { fprintf(stderr, "Use: %s <n>\n", argv[0]); return 1; }

  n = atoi(argv[1]);
  for (i = 1; i < n; i++)
    if (childpid = fork())
      break;
   fprintf(stderr,"i:%d pid:%ld ppid:%ld child:%ld\n",
       i, (long)getpid(), (long)getppid(), (long)childpid);
  return 0;
}
```

USP Program 3.2 — A Fan of Processes (modified)

```
#include <stdio.h> <stdlib.h> <unistd.h>

int main (int argc, char ∗argv[]) { /* simplefan.c */
  pid_t childpid = 0;
  int i, n;
   if (argc ≠ 2)  /* check for no. of cmd-line args */
   { fprintf(stderr, "Use: %s <n>\n", argv[0]); return 1; }

  n = atoi(argv[1]);
  for (i = 1; i < n; i++)
    if ((childpid = fork()) ≤ 0)
      break;
   fprintf(stderr,"i:%d pid:%ld ppid:%ld child:%ld\n",
       i, (long)getpid(), (long)getppid(), (long)childpid);
  return 0;
}
```

USP Exercise 3.10 — A Tree of Processes (modified)

```
#include <stdio.h> <stdlib.h> <unistd.h> /* simpletree.c */

int main (int argc, char *argv[]) {
  pid_t childpid = 0;
  int i, n;
  if (argc ≠ 2)  /* check for valid command-line */
   { fprintf(stderr, "Usage: %s processes\n", argv[0]); return 1; }

  n = atoi(argv[1]);
  for (i = 1; i < n; i++)
    if ((childpid = fork()) == –1)      /* the only change */
      break;
  fprintf(stderr, "i:%d  process: %ld  parent: %ld  child: %ld\n",
        i, (long)getpid(), (long)getppid(), (long)childpid);
  return 0;
}
```

wait

wait: the calling process waits until one of its children exits — this is a special kind of **process synchronisation**

- **wait** returns PID of exited child to caller
- can also return exit status of child
- including whether child was terminated by a signal, and by which signal

waitpid is more general:

- can wait for specified child, or for children from specified process group
- an option makes it **non-blocking**
- can also report stopped children

Interruption by Signals

- **Signals** are a special kind of IPC "messages"
- A process can install **signal handlers**
- (Most) unhandled signals terminate the process
- Signals can arrive at any time
- Signals can interrupt certain system calls — *EINTR*

```
#include <errno.h>                          /* r_wait.c */
#include <sys/wait.h>

pid_t r_wait(int *stat_loc) {
  int retval;

  while (((retval = wait(stat_loc)) == –1) && (errno == EINTR)) ;
  return retval;
}
```

USP Program 3.4 — forking off "`ls -l`"

```
#include <stdio.h> <stdlib.h> <unistd.h> <sys/wait.h>
                                              /* execls.c */
int  main(void) {
  pid_t childpid;
  childpid = fork();
  if (childpid == –1)  { perror("Failed to fork"); return 1; }
  if (childpid == 0) {                        /* child code */
     execl("/bin/ls", "ls", "-l", NULL);
     perror("Child failed to exec ls"); return 1;
  }
  if (childpid ≠ wait(NULL)) {                /* parent code */
     perror("Parent failed to wait due to signal or error");
     return 1;
  }
  return 0;
}
```

USP Program 3.5 — delegating a command

```
#include <errno.h> <stdio.h> <unistd.h> <sys/...>
#include "restart.h"                         /* execcmd.c */
int main(int argc, char *argv[]) {
  pid_t childpid;
  if (argc < 2) { /* check for valid number of cmd-line args */
    fprintf (stderr, "Usage: %s cmd arg1 arg2 ...\n", argv[0]);
    return 1; }
  childpid = fork();
  if (childpid == –1) { perror("Failed to fork"); return 1; }
  if (childpid == 0) {                           /* child code */
    execvp(argv[1], &argv[1]);
    perror("Child failed to execvp the command"); return 1; }
  if (childpid ≠ r_wait(NULL))                   /* parent code */
   { perror("Parent failed to wait"); return 1; }
  return 0;
}
```

_exit(), *exit*(), and exit handlers

- void *_exit*(int *status*): system call, terminates the calling process regularly, with exit status *status* (in unistd.h).
- void *exit*(int *status*): library function, calls user-defined exit handlers, performs cleanup, then calls *_exit*(*status*) (in stdlib.h).
- int *atexit*(void (*f)(void))
  installs function *f* as an exit handler.

  If several handlers are installed, they are called in *reverse* installation order.
- return *k*; in *main* is equivalent to exit(k);

USP Program 3.7 — creating a background process

```
int makeargv(const char *, const char *, char ***);
int main(int argc, char *argv[]) {                    /* runback.c */
  pid_t childpid; char delim[] = " \t"; char **myargv;
  if (argc ≠ 2) { fprintf(stderr, "Usage: ..."); return 1; }
  childpid = fork();
  if (childpid == –1) { perror("Failed to fork"); return 1; }
  if (childpid == 0) {  /* child becomes a background process */
   if (setsid() == –1) perror("failed to become session leader");
   else if (makeargv(argv[1], delim, &myargv) == –1)
     fprintf(stderr, "Child failed to construct arg. array\n");
   else
     { execvp(myargv[0], &myargv[0]); perror("exec failed"); }
   return 1;                        /* child should never return */
  }
  return 0;                                        /* parent exits */
}
```

USP Program 2.10 — Example Exit Handling Function

```
static void show_times(void)                     /* showtimes.c (1) */
{  struct tms times_info; double ticks;
  if ((ticks = (double) sysconf(_SC_CLK_TCK)) < 0)
    perror("Cannot determine clock ticks per second");
  else if (times(&times_info) < 0)
    perror("Cannot get times information");
  else {
    fprintf(stderr, "User time:                %8.3f seconds\n",
      times_info.tms_utime/ticks);
    fprintf(stderr, "System time:              %8.3f seconds\n",
      times_info.tms_stime/ticks);
    fprintf(stderr, "Children's user time:   %8.3f seconds\n",
      times_info.tms_cutime/ticks);
    fprintf(stderr, "Children's system time: %8.3f seconds\n",
      times_info.tms_cstime/ticks);
}}
```

USP Program 2.10 (ctd.) — Exit Handler Example

```
void show_times(void);                    /* showtimes.c (2) */

void main(void)
{  if (atexit(show_times))  {
       fprintf(stderr, "Cannot install show_times exit handler\n");
       return 1;
    }

    /*  rest of main program goes here */

   return 0;
}
```