Processes

Slides adapted from: Randy Bryant of Carnegie Mellon University
Processes

Definition: A process is an instance of a running program.
- One of the most profound ideas in computer science
- Not the same as “program” or “processor”

Process provides each program with two key abstractions:
- Logical control flow
  - Each program seems to have exclusive use of the CPU
- Private virtual address space
  - Each program seems to have exclusive use of main memory

How are these illusions maintained?
- Process executions interleaved (multitasking)
- Address spaces managed by virtual memory system
What is a process?

- A process is the OS's abstraction for execution
  - A process represents a single running application on the system
- Process has three main components:
  1. **Address space**
     - The memory that the process can access
     - Consists of various pieces: the program code, static variables, heap, stack, etc.
  2. **Processor state**
     - The CPU registers associated with the running process
     - Includes general purpose registers, program counter, stack pointer, etc.
  3. **OS resources**
     - Various OS state associated with the process
     - Examples: open files, network sockets, etc.
Virtual memory of a process includes:

- the code of the running program
- the data of the running program (static variables and heap)
- the execution stack storing local variables and saved registers for each procedure call
This is the process's own view of the address space

- Physical memory may not be laid out this way at all.
- The virtual memory system provides this illusion to each process.
Execution State (context) of a Process

- Each process has an execution state (context)
  - Indicates what the process is currently doing

- Running:
  - Process is currently using the CPU

- Ready:
  - Currently waiting to be assigned to a CPU
  - That is, the process could be running, but another process is using the CPU

- Waiting (or sleeping):
  - Process is waiting for an event
  - Such as completion of an I/O, a timer to go off, etc.
  - Why is this different than “ready”?

- As the process executes, it moves between these states
  - What state is the process in most of the time?
Process State (Context) Transitions

- What causes schedule and unschedule transitions?

```
New -> Ready
  ^              ^
  | create       | unschedule
Terminated <- Running
  | schedule     | I/O done
Waiting

Running <- Terminated
  | kill or exit |
```

I/O, page fault, etc.
Process Control Block

- OS maintains a Process Control Block (PCB) for each process
- The PCB is a big data structure with many fields:
  - Process ID
  - User ID
  - Execution state
    - ready, running, or waiting
  - Saved CPU state
    - CPU registers saved the last time the process was suspended.
  - OS resources
    - Open files, network sockets, etc.
  - Memory management info
  - Scheduling priority
    - Give some processes higher priority than others
  - Accounting information
    - Total CPU time, memory usage, etc.
PCB in Linux

Each **task_struct** data structure describes a process or task in the system.
Context Switching

- Processes are managed by a shared chunk of OS code called the **kernel**
  - Important: the kernel is not a separate process, but rather runs as part of some user process

- **Control flow passes from one process to another via a context switch**
Context Switching in Linux

Process A is happily running along...
Context Switching in Linux

1) Timer interrupt fires
2) PC saved on stack
Context Switching in Linux

1) Timer interrupt fires
2) PC saved on stack
3) Rest of CPU state saved in PCB
4) Call schedule() routine

Process A

Timer interrupt handler

Scheduler
Context Switching in Linux

1) Timer interrupt fires
2) PC saved on stack
3) Rest of CPU state saved in PCB
4) Call schedule() routine
5) Decide next process to run
6) Resume Process B (suspended within timer interrupt handler!)
7) Return from interrupt handler – process CPU state restored
Context Switch Overhead

- **Context switches are not cheap**
  - Generally have a lot of CPU state to save and restore
  - Also must update various flags in the PCB
  - Picking the next process to run – scheduling – is also expensive

- **Context switch overhead in Linux**
  - About 5-7 usec (u: micro)
  - This is equivalent to about 10,000 CPU cycles!
The OS maintains a set of state queues for each process state
- Separate queues for ready and waiting states
- Generally separate queues for each kind of waiting process
  - One queue for processes waiting for disk I/O, another for network I/O, etc.
State Queue Transitions

- PCBs move between these queues as their state changes
  - When scheduling a process, pop the head off of the ready queue
  - When I/O has completed, move PCB from waiting queue to ready queue
Concurrent Processes

- Each process is a logical control flow.
- Two processes run *concurrently* (are concurrent) if their flows overlap in time.
- Otherwise, they are *sequential*.
- Examples (running on single core):
  - Concurrent: A & B, A & C
  - Sequential: B & C

![Diagram showing concurrent processes](image-url)
User View of Concurrent Processes

- Control flows for concurrent processes are physically disjoint in time

- However, we can think of concurrent processes as running in parallel with each other
Creating Processes

- *Parent process* creates a new running *child process* by calling *fork*.
Creating Processes

- *Parent process* creates a new running *child process* by calling *fork*

- `int fork(void)`
  - Returns 0 to the child process, child’s PID to parent process
  - Child is *almost* identical to parent:
    - Child get an identical (but separate) copy of the parent’s virtual address space.
    - Child gets identical copies of the parent’s open file descriptors
    - Child has a different PID than the parent

- *fork* is interesting (and often confusing) because it is called *once* but returns *twice*

  [https://www.cdn.geeksforgeeks.org/fork-system-call/](https://www.cdn.geeksforgeeks.org/fork-system-call/)
fork Example

```c
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = fork();
    if (pid == 0) { /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```

- Call once, return twice
- Concurrent execution
  - Can’t predict execution order of parent and child
fork Example

```c
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = fork();
    if (pid == 0) { /* Child */
        printf("child: x=%d\n", ++x);
        printf("child: x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    printf("parent: x=%d\n", --x);
    return 0;
}
```

- Call once, return twice
- Concurrent execution
  - Can’t predict execution order of parent and child
- Duplicate but separate address space
  - x has a value of 1 when fork returns in parent and child
  - Subsequent changes to x are independent

```bash
linux> ./fork
parent: x=0
child : x=2
parent: x=-1
child : x=3
```
**fork Example**

```c
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = fork();
    if (pid == 0) { /* Child */
        printf("child : x=%d\n", ++x);
        /* printf("child : x=%d\n", ++x); */
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    printf("parent: x=%d\n", --x);
    return 0;
}
```

- Call once, return twice
- Concurrent execution
  - Can’t predict execution order of parent and child
- Duplicate but separate address space
  - x has a value of 1 when fork returns in parent and child
  - Subsequent changes to x are independent
- Shared open files
  - stdout is the same in both parent and child
Modeling `fork` with Process Graphs

- A *process graph* is a useful tool for capturing the partial ordering of statements in a concurrent program:
  - Each vertex is the execution of a statement
  - $a \rightarrow b$ means $a$ happens before $b$
  - Edges can be labeled with current value of variables
  - `printf` vertices can be labeled with output
  - Each graph begins with a vertex with no in-edges

- Any *topological sort* of the graph corresponds to a feasible total ordering.
  - Total ordering of vertices where all edges point from left to right
```c
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = fork();
    if (pid == 0) { /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }
    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```

`fork.c`
Interpreting Process Graphs

- **Original graph:**

- **Relabeled graph:**

- **Feasible total ordering:**

- **Infeasible total ordering:**
fork Example: Two consecutive forks

```c
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

Feasible output:
L0
L1
Bye
Bye
Bye

Infeasible output:
L0
Bye
L1
Bye
L1
Bye
Bye
Bye

fork Example: Nested forks in parent

```c
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

Feasible output:
```
L0
L1
L2
Bye
```

Infeasible output:
```
L0
Bye
L1
Bye
L2
Bye
```
void fork5() {
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }        
    }
    printf("Bye\n");
}

Feasible output:    Infeasible output:
L0                    L0
Bye                   Bye
L1                    L1
L2                    Bye
Bye                   Bye
Bye                   L2
Why have fork() at all?

- Why make a copy of the parent process?
- Don't you usually want to start a new program instead?
- Where might “cloning” the parent be useful?
  - Web server – make a copy for each incoming connection
  - Parallel processing – set up initial state, fork off multiple copies to do work

- UNIX philosophy: System calls should be minimal.
  - Don't overload system calls with extra functionality if it is not always needed.
  - Better to provide a flexible set of simple primitives and let programmers combine them in useful ways.
What if *fork*’ing gets out of control?

```c
void forkbomb() {
    while (1)
        fork();
}
```
Memory concerns

- OS aggressively tries to share memory between processes.
  - Especially processes that are `fork()`'d copies of each other

- Copies of a parent process do not actually get a private copy of the address space...
  - ... though that is the illusion that each process gets.
  - Instead, they share the same physical memory, until one of them makes a change.

- The virtual memory system is behind these tricks.
  - We will discuss this in much detail later in the course
Terminating Processes

- **Process becomes terminated for one of three reasons:**
  - Receiving a signal whose default action is to terminate (more later)
  - Returning from the `main` routine
  - Calling the `exit` function

- **void exit(int status)**
  - Terminates with an *exit status* of *status*
  - Convention: normal return status is 0, nonzero on error
  - Another way to explicitly set the exit status is to return an integer value from the main routine

- exit is called once but never returns.

```c
void cleanup(void) {
    printf("cleaning up\n");
}

void fork() {
    atexit(cleanup);
    fork();
    exit(0);
}
```
Reaping Child Processes

- **Idea**
  - When process terminates, it still consumes system resources
    - Examples: Exit status, various OS tables
  - Called a “zombie”
    - Living corpse, half alive and half dead

- **Reaping**
  - Performed by parent on terminated child (using `wait` or `waitpid`)
    - Parent is given exit status information
    - Kernel then deletes zombie child process

- **What if parent doesn’t reap?**
  - If any parent terminates without reaping a child, then the orphaned child will be reaped by `init` process (pid == 1)
  - So, only need explicit reaping in long-running processes
    - e.g., shells and servers
Zombie Example

```c
void fork7() {
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n", getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

- `ps` shows child process as "defunct" (i.e., a zombie)
- Killing parent allows child to be reaped by `init`
Non-terminating Child Example

```c
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n", getpid());
        exit(0);
    }
}
```

Child process still active even though parent has terminated

Must kill child explicitly, or else will keep running indefinitely
wait: Synchronizing with Children

- Parent reaps a child by calling the `wait` function

- `int wait(int *child_status)`
  -Suspends current process until one of its children terminates

---

**Parent Process**  

<table>
<thead>
<tr>
<th>syscall...</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns</td>
<td>-----------</td>
</tr>
</tbody>
</table>

And, potentially other user processes, including a child of parent
**wait**: Synchronizing with Children

- Parent reaps a child by calling the `wait` function

```c
int wait(int *child_status)
```

- Suspends current process until one of its children terminates
- Return value is the **pid** of the child process that terminated
- If `child_status != NULL`, then the integer it points to will be set to a value that indicates reason the child terminated and the exit status:
  - Checked using macros defined in `wait.h`
    - `WIFEXITED`, `WEXITSTATUS`, `WIFSIGNALED`, `WTERMSIG`, `WIFSTOPPED`, `WSTOPSIG`, `WIFCONTINUED`
    - See man pages for details
Process completion status

- **int WIFEXITED (int status)**
  - returns a nonzero value if the child process terminated normally with exit or _exit.

- **int WEXITSTATUS (int status)**
  - If WIFEXITED is true of status, this macro returns the low-order 8 bits of the exit status value from the child process.

- **int WIFSIGNALED (int status)**
  - returns a nonzero value if the child process terminated because it received a signal that was not handled

- **int WTERMSIG (int status)**
  - If WIFSIGNALED is true of status, this macro returns the signal number of the signal that terminated the child process.

- **int WCOREDUMP (int status)**
  - Returns a nonzero value if the child process terminated and produced a core dump.

- **int WIFSTOPPED (int status)**
  - returns a nonzero value if the child process is stopped.

- **int WSTOPSIG (int status)**
  - If WIFSTOPPED is true of status, this macro returns the signal number of the signal that caused the child process to stop.

wait: Synchronizing with Children

```c
void fork9() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
        exit(0);
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}
```

Feasible output(s):

<table>
<thead>
<tr>
<th>HC</th>
<th>HP</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP</td>
<td>HC</td>
</tr>
<tr>
<td>CT</td>
<td>CT</td>
</tr>
<tr>
<td>Bye</td>
<td>Bye</td>
</tr>
</tbody>
</table>

Infeasible output:

<table>
<thead>
<tr>
<th>HP</th>
</tr>
</thead>
<tbody>
<tr>
<td>CT</td>
</tr>
<tr>
<td>Bye</td>
</tr>
<tr>
<td>HC</td>
</tr>
</tbody>
</table>
Another wait Example

- If multiple children completed, will take in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```c
void fork10() {
    pid_t pid[N];
    int i, child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            exit(100+i); /* Child */
        }

    for (i = 0; i < N; i++) {
        /* Parent */
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```

forks.c
waitpid: Waiting for a Specific Process

- **pid_t waitpid(pid_t pid, int *status, int options)**
  - Suspends current process until specific process terminates
  - Various options (see man page)

```c
void fork11() {
    pid_t pid[N];
    int i;
    int child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = N-1; i >= 0; i--)
        { pid_t wpid = waitpid(pid[i], &child_status, 0);
          if (WIFEXITED(child_status))
              printf("Child %d terminated with exit status %d\n",
                      wpid, WEXITSTATUS(child_status));
          else
              printf("Child %d terminate abnormally\n", wpid);
        }
}
```
**execve: Loading and Running Programs**

- **int execve(char *filename, char *argv[], char *envp[])**
- **Loads and runs in the current process:**
  - Executable file **filename**
    - Can be object file or script file beginning with `#!/interpreter` (e.g., `#!/bin/bash`)
  - ...with argument list **argv**
    - By convention `argv[0] == filename`
  - ...and environment variable list **envp**
    - "name=value" strings (e.g., `USER=droh`)
    - `getenv`, `putenv`, `printenv`
- **Overwrites code, data, and stack**
  - Retains PID, open files and signal context
- **Called once and never returns**
  - ...except if there is an error
fork() and execve()

- execve() does not fork a new process!
  - Rather, it replaces the address space and CPU state of the current process
  - Loads the new address space from the executable file and starts it from main()
  - So, to start a new program, use fork() followed by execve()
**execl and exec Family**

- int execl(char *path, char *arg0, char *arg1, ..., 0)
- **Loads and runs executable at path with args arg0, arg1, ...**
  - *path* is the complete path of an executable object file
  - By convention, *arg0* is the name of the executable object file
  - “Real” arguments to the program start with *arg1*, etc.
  - List of *args* is terminated by a (*char *) 0 argument
  - Environment taken from *char **environ*, which points to an array of “name=value” strings:
    - USER=ganger
    - LOGNAME=ganger
    - HOME=/afs/cs.cmu.edu/user/ganger
- **Returns −1 if error, otherwise doesn’t return!**
- Family of functions includes execv, execve (base function), execvp, execl, execle, and execlp
exec: Using fork followed by exec

```c
int main(int argc, char **argv) {
    int rv;
    if (fork()) { /* Parent process */
        wait(&rv);
    } else { /* Child process */
        char *newargs[3];
        printf("Hello, I am the child process.\n");
        newargs[0] = "/bin/echo"; /* Convention! Not required!! */
        newargs[1] = "some random string";
        newargs[2] = NULL; /* Indicate end of args array */
        if (execv("/bin/echo", newargs)) {
            printf("warning: execve returned an error.\n"); exit(-1);
        }
        printf("Child process should never get here\n");
        exit(42);
    }
}
```
exec() function family

- The suffix’s determine the arguments
  - `l`: arguments are passed as a list of strings to the main()
  - `v`: arguments are passed as an array of strings to the main()
  - `p`: path/s to search for the new running program
  - `e`: the environment can be specified by the caller

- One can mix them with different combinations
  - `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 execve(const char *path, char *const argv[], char *const envp[])`;
  - `int execvp(const char *file, char *const argv[])`;
  - `int execvpe(const char *file, char *const argv[], char *const envp[])`;

- The initial argument is always the name of a file to be executed.
Environment variables

- An environment variable is a dynamic-named value that can affect the way running processes will behave on a computer.
- They are part of the environment in which a process runs. For example,
  - a running process can query the value of the TEMP environment variable to discover a suitable location to store temporary files,
  - or the HOME or USERPROFILE variable to find the directory structure owned by the user running the process
- In Unix, use `printenv` in your shell to get the full list.

Linux Process Hierarchy

Note: you can view the hierarchy using the Linux `pstree` command
Summary

- **Process**
  - is an instance of program in execution
  - At any given time, a system has multiple active processes
  - Only one can execute at a time, though
  - Each process appears to have total control of processor + private memory space

- **Spawning processes**
  - Call to `fork`
  - One call, two returns

- **Process completion**
  - Call `exit`
  - One call, no return

- **Reaping and waiting for Processes**
  - Call `wait` or `waitpid`

- **Loading and running Programs**
  - Call `exec1` (or variant)
  - One call, (normally) no return