Processes - A System View
Waiting for a child to finish – \texttt{wait()}

\begin{verbatim}
#include <sys/types.h>
#include <wait.h>

pid_t wait(int *status);
\end{verbatim}

- Suspend calling process until child has finished
- Allow parent to reap child
- Returns:
  - Process ID of terminated child on success
  - -1 on error, sets \texttt{errno}
- Parameters:
  - \texttt{status}: status information set by \texttt{wait} and evaluated using specific macros defined for \texttt{wait}.
Waiting for a child to finish –

`wait()`

```
#include <sys/types.h>
#include <wait.h>
pid_t wait(int *status);
```

- Suspend calling process until child has finished
- Allow parent to reap child
- Returns:
  - Process ID of terminated child on success
  - -1 on error, sets `errno`
- Parameters:
  - `status`: status information set by `wait` and evaluated using specific macros defined for `wait`.

Instead of waiting, you can use a signal handler (later lecture) for signal `SIGCHLD` which issues a `wait()` call.
wait() syscall

- Allows parent process to wait (block) until child finishes
- Causes the caller to suspend execution until child’s status is available

<table>
<thead>
<tr>
<th>errno</th>
<th>cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECHILD</td>
<td>Caller has no unwaited-for children</td>
</tr>
<tr>
<td>EINTR</td>
<td>Function was interrupted by signal</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Options parameter of waitpid was invalid</td>
</tr>
</tbody>
</table>
void fork9() {
    int child_status;

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

```c
int execv(char *filename, char *argv[])
```

- transforms the calling process into a new process
  - Runs executable `filename`
  - With argument list `argv`
- Does not return (unless error)
- Overwrites code, data, and stack
  - keeps pid, open files and signal context
**execv**: Loading and Running Programs

```c
int execv(char *filename, char *argv[])
```

- `argv` is a pointer to the argument list to be made available to the new process

To pass arguments and environment, use:

```c
int execve(char *filename, char *argv[], char *envp[])
```
```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    int fd;

    fd = open(argv[1],O_RDWR|O_CREAT,S_IRWXU);    // ???
dup2(fd, 1);                                   // ???
close(fd);                                     // ???

    char* array[] = {"ls", "-la", NULL};
    execv("/bin/ls", array);

    printf("This string should not be printed!\n");
}
```
```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    int fd;

    fd = open(argv[1], O_RDWR|O_CREAT, S_IRWXU); //create an output file
    dup2(fd, 1); //redirect output to file
    close(fd); //free unused file descriptor

    char* array[] = {"ls", "-la", NULL};
    execv("/bin/ls", array);

    printf("This string should not be printed!\n");
}
```
Concurrent Processes

- 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
What is fork good for?

- What does concurrency gain us?
  - The appearance that multiple actions are occurring at the same time
  - If done right, your program can improve throughput (#instr./second)

- `fork()` creates a new process that runs concurrently
What is fork good for?

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
    pid_t pid;
    int i;

    if (pid = fork()) { /* parent */
        parentProcedures();
    }
    else { /* child */
        childProcedures();
    }

    return 0;
}
```
What is fork good for?

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
    pid_t pid;
    int i;
    while (1) {
        /* wait for new clients */
        if(pid = fork()) { /* parent */
            /* reset server */
        }
        else { /* child */
            /* handle new client */
            exit(0);
        }
    }
    return 0;
}
```

Copyright ©: University of Illinois CS 241 Staff
Why Concurrency?

- Exploit natural concurrent structure of an application
  - The world is not sequential!
  - Easier to program multiple independent and concurrent activities

- Better resource utilization
  - Resources unused by one application can be used by the others

- Better average response time
  - No need to wait for other applications to make progress
Benefits of Concurrency

<table>
<thead>
<tr>
<th>No Concurrency</th>
<th>With Concurrency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client 1</td>
<td>Client 1</td>
</tr>
<tr>
<td>Client 2</td>
<td>Client 2</td>
</tr>
<tr>
<td>Client 3</td>
<td>Client 3</td>
</tr>
</tbody>
</table>

**Wait for input**

Copyright ©: University of Illinois CS 241 Staff
Five State Process Model

- New
  - New process is created

- Ready
  - Available to execute

- Running
  - Currently executing
  - On a single processor machine, at most one process in the “running” state

- Blocked
  - Waiting on some event

- Done
  - Process terminates
5 State Model - Transitions

- New process creation

Diagram:
- new
- ready
- running
- blocked
- done
- enter

Copyright ©: University of Illinois CS 241 Staff
5 State Model - Transitions

- New to Ready
  - Move to pool of ready processes

States:
- New
- Ready
- Running
- Blocked
- Done
5 State Model - Transitions

- Ready for Running
  - Chosen to run from the pool of ready processes (How?)

States:
- new
- ready
- running
- blocked
- done
5 State Model - Transitions

What events cause these transitions?
5 State Model - Transitions

- Running to Ready
  - Preempted

- Running to Blocked
  - Request for an unavailable resource

- Running to Done
  - Terminated by the OS
5 State Model - Transitions

- Blocked to Ready
  - Resource is now available
5 State Model - Transitions

- Ready to Done
  - Involuntary termination
- Blocked to Done
  - Involuntary termination
5 State Model - Transitions

- **new**: process created
- **ready**: selected to run
- **running**: quantum expired
- **blocked**: I/O request (syscall)
- **done**: normal or abnormal termination

- **I/O complete**: transitions to blocked
- **terminated**: transitions to done